Supporting consistency in linked specialized engineering models through bindings and updating.

A.H. Olivier

Dissertation presented for the degree of Doctor of Philosophy at
Stellenbosch University

Study leaders:
Dr. G.C. van Rooyen
Prof. K.E. Beucke
Prof. B. Firmenich

December 2007

Declaration

I, the undersigned, hereby declare that the work contained in this dissertation is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Ek, die ondergetekende, verklaar hiermee dat die werk in hierdie proefskrif vervat, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik by enige universiteit ter verkryging van 'n graad voorgelê het nie.

Signature: Date: 11 November 2007

Synopsis

Elements of the Architecture-Engineering-Construction (AEC) industry are the result of a process involving planning, engineering and construction. A number of professions and professionals are involved, and the process is characterized by frequent changes. Consequently the problem of consistency of information is a major concern which casts a shadow on the integrity of the process. The research described in this dissertation was aimed at the development of techniques and technologies which can alleviate the problem of information exchange and consistency.

Currently some commercial software applications support users working in an integrated environment in the exchange of information between different models. However, this is limited to the suite of models provided by the software vendor and consequently it forces all the parties involved in a project to use the same software. This excludes potential participants and the software-suites are usually expensive as well.

In contrast, the research described here investigated ways of using standard software applications, which may be specialized for different professional domains. These are linked for effective transfer of information and a binding mechanism is provided to support consistency between the models. This prevents the exclusion of participants, allowing them to use familiar software packages, without losing the ability to keep the various models consistent amongst project partners. This is of particular importance to specialists that use problem specific applications which may not be included in expensive, integrated suites.

The solution approach presented in the dissertation accounts for the following well known properties of the AEC industry:

Ownership - each model that abstracts a specific aspect of the project is created, manipulated and controlled by a responsible person/party. No one may circumvent the model owner to manipulate a model.

Diversity - the various role players often do not understand the complexity and value of the work of the other parties involved.

Long transactions - the time duration of tasks in the construction industry is not short and information cannot be managed on a transaction basis. The various role players must be supported to work in parallel, exchanging relevant information constantly as the project develops.

The proposed solution consists of a linking and binding mechanism that supports the definition of inter-object dependencies. These dependencies are described by `Binder` instances. `Update` behavior is assigned to `Binder` instances through customized `Updater` instances. The binding mechanism addresses important issues like change detection, update sequence determination and the execution of an update in dependent models.

The proposed solution was successfully implemented using a CAD system and an independent Finite Element application in order to verify the theoretical aspects of the work.

Opsomming

Elemente van die Argitektuur-Ingenieurswese-Konstruksie (AIK) industrie is die resultaat van 'n proses bestaande uit beplanning, ingenieurswese en konstruksie. 'n Aantal professies en professionele persone is betrokke en die proses word gekenmerk deur gereelde veranderinge. Gevolglik is die probleem van die konsekwentheid van inligting van kardinale belang aangesien die gebruik van verouderde inligting die integriteit van die proses kan benadeel. In hierdie verhandeling word navorsing beskryf wat gerig is op die ontwikkeling van tegnieke en tegnologieë wat die probleme geassosieër met die oordra en konsekwentheid van inligting kan verlig.

Sekere kommersiële rekenaartoepassings is beskikbaar wat gebruikers in 'n geïntegreerde omgewing ondersteun in die uitruil van inligting tussen verskillende modelle. Dit is egter beperk tot die modelle wat deur die sagteware ontwikkelaar self verskaf word en bied dus net 'n oplossing indien al die partye betrokke by 'n projek dieselfde sagteware gebruik. Dit kan daartoe aanleiding gee dat potensiële deelnemers van die proses uitgesluit word. Die sagteware pakette is gewoonlik ook duur.

Hierteenoor het die navorsing hier beskryf maniere ondersoek om algemene sagteware-toepassings, wat gespesialiseer kan wees vir verskillende professionele domeine, te gebruik. Hierdie toepassings is gekoppel vir die effektiewe oordrag van inligting en 'n bindmeganisme word verskaf om konsekwentheid tussen modelle te verseker. Die uitsluiting van deelnemers word sodoende verhoed. Hulle word ook toegelaat om bekende sagteware pakette te gebruik, sonder om die vermoeë in te boet om die verskeie modelle tussen deelnemers konsekwent te hou. Dit is spesifiek belangrik vir spesialiste wat spesifieke toepassings gebruik wat nie noodwendig ingesluit is in duur, geïntegreerde pakette nie.

Die benadering tot die oplossing van die probleem wat in hierdie verhandeling gevolg is neem die volgende bekende einskappe van die AIK industrie in ag:

Eienaarskap - elke model wat 'n spesifieke aspek van 'n projek verteenwoordig word geskep, gemanipuleer en beheer deur 'n verantwoordelike persoon/party. Niemand anders mag die model verander nie.

Diversiteit - die verskillende rolspelers verstaan nie noodwendig die kompleksiteit en waarde van die werk van die ander betrokke partye nie.

Lang transaksies - die tydsduur van take in die konstruksie industrie is nie kort nie en inligting kan nie op 'n transaksie grondslag bestuur word nie. Die verskillende rolspelers moet ondersteun word om parallel te kan werk terwyl inligting voortdurend uitgeruil word soos die projek ontvou.

Die voorgestelde oplossing bestaan uit 'n koppel- en bindmeganisme wat die definiëring van inter-objek afhanklikhede ondersteun. Hierdie afhanklikhede word beskryf deur objekte van klas `Binder`. Veranderinge word hanteer deur doelgemaakte `Updater` objekte wat aan die `Binder` objekte toegeken word. Die bindmeganisme spreek belangrike kwessies aan, byvoorbeeld die waarneming van verandering, bepaling van die volgorde waarin veranderinge aangebring moet word, asook die aanbring van veranderinge in afhanklike modelle.

Die voorgestelde oplossing is suksesvol implementeer deur die koppeling van 'n rekenaar gesteunde ontwerp (CAD) stelsel en 'n onafhanklike Eindige Element toepassing, waardeur die teoretiese aspekte van die werk bevestig kon word.

Acknowledgements

Looking back at the last five years of my life, I would like to take this opportunity to thank the persons and institutions that made this dissertation possible.

The first person that comes to mind is my wife, Yolandé. She supported me all the way, she never complained about the things that she had to sacrifice to enable me to focus on this dissertation. Thank you Yolandé! I also would like to thank the rest of my family (especially my parents and parents-in-law) for their support during this time, for their visits to Germany and all the prayers said on our behalf.

The second person that played a major role is my promoter and dear friend, Dr. G.C. van Rooyen. Thank you for your support and guidance during my post graduate studies.

Further I would like to thank my two German study leaders who were always available when I needed help and mentoring in Germany: Prof. K.E. Beucke, for assisting me and my family in moving to Germany and supporting us and Prof. B. Firmenich, for his dedication and encouragement during the search for software techniques to solve the problem described in this dissertation.

Fourthly, my dear friend Riaan Burger, who was always available for an independent opinion and also for proof reading this document. Thanks for your friendship and support, especially throughout the last five years.

Then also thank you to the co-workers at the Bauhaus-Universität Weimar, especially Eike Tauscher, Christian Koch and Torsten Richter. You guys not only gave me valuable input on this dissertation, but were also real friends in a foreign country.

Finally, I would like to thank the following institutions for financial support: the Deutscher Akademisher Austausch Dienst (DAAD) for supporting me and my family during the two years that we have spent in Germany. I am also grateful to the Southern African Steel Institute for their support during the first part of my studies in South Africa. Without the support of these two organisations, this dissertation would not have been possible.

vir Yolandé

"To him who is able to keep you from falling and to present you before his glorious presence without fault and with great joy – to the only God our Savior be glory, majesty, power and authority, through Jesus Christ our Lord, before all ages, now and forevermore! Amen." Jude 24-25

Contents

Chapter 1

Introduction

1.1 Problem statement

Elements of the built environment are the result of a process involving planning, engineering and construction. A number of professions and professionals are involved, and the process is characterised by frequent changes. Consequently the problem of consistency of information is a major concern which casts a shadow on the integrity of the process. The research described in this dissertation is aimed at the development of techniques and technologies which can alleviate the problem of information consistency and the specific focus is on the structural design task.

Structural design directly depends on the geometric properties of the entity under consideration, where the geometric properties are generally expressed in Computer-Aided Design (CAD) models. If the geometric properties change the dependent structural design becomes inconsistent and needs to be adapted to the new geometry. Since this type of dependency is typical of many engineering tasks, it is expected that the results of the research will apply to a broad class of engineering work.

1.2 The construction industry

The construction industry requires several different role players to work together to achieve a common goal, namely the erection of a new building. As a result three aspects of the construction industry must be recognized and supported when new technologies are introduced to enhance the industry, namely ownership, the diverse nature of the role players and the fact that engineering transactions are long. Each of these aspects is briefly described below.

1.2.1 Ownership

The architecture, engineering and construction process (AEC) is divided into different tasks, each of which creates and/or modifies datasets as the building evolves. Each task has a responsible person or group assigned to it who uses information obtained elsewhere to execute the task under consideration. Specialized knowledge and experience are used to 'translate' the information on which the task is based to task specific information. For example, a structural engineer receives a floor layout plan from an architect and has to perform a structural design of the floor slab. This requires an interpretation of the geometry and annotations to devise a load-bearing mechanism for the floor. Load values must be assigned and these require an interpretation of the function of the room, e.g. the imposed load in a factory is much higher than that of a residential apartment.

The architect assigns the function of a room and based on that, the structural engineer assigns the design loads. If the room function changes, the load assignment needs to be revised. In this example the architect is responsible for assigning room functions and the structural engineer is responsible for mapping the room function to design loads. The architect cannot be allowed to change the design loads without the knowledge and approval of the structural engineer because this may compromise the slab design.

Each role player is responsible for his/her decisions and for the information that stems from those decisions. Consequently each responsible person must be completely in control of his information. No role player should adjust information that belongs to a different role player.

1.2.2 Diversity

In the construction industry the various role players often do not understand the complexity and value of the work of the other parties involved. Each role player has a different view of some building component. An architect, for example, does not think in terms of bending moments while drawing a beam or floor slab. Likewise, the structural engineer does not think about geometric proportions while considering the same objects. This exacerbates the consistency problem, since the different disciplines depend on each other and have to work together to reach the final product.

1.2.3 Long transactions

In many sectors of industry access to information can be locked while that information is used by a task, releasing the lock as soon as the task is completed. This has the advantage that information can be kept consistent since only one task or party can access the information at any given time. Transaction-based control over information, however, can only be applied

in an environment where the transactions are short. The time duration of tasks in the construction industry is not short and information cannot be managed on a transaction basis. The various role players must be supported to work in parallel, exchanging relevant information constantly as the project develops.

1.3 Sharing of information

The way in which information sharing is supported by computer technology has a profound influence on software techniques which are proposed to alleviate information consistency. Essentially two approaches are possible, namely sharing by reference and sharing by value.

1.3.1 Information sharing by reference

The sharing of information by reference means that the information contents reside at a certain location, and that the address of this location is shared amongst participants. No duplication of the information occurs i.e. no redundant information exists in such a system e.g. a relational database that is normalized [Date 2000]. Different tasks act on the same information contents and modify it as time goes by. If all information is shared by reference consistency problems cannot occur. Consider, for example, a beam in an architectural model with length l. Inside the structural engineering model the beam is reached by reference and the length of the beam is always obtained using the reference when needed. Consequently the length of the beam inside the structural engineering model will always be consistent with the length of the beam inside the architectural model.

The disadvantage of sharing information by reference is that a model which references information residing in another model cannot function on its own anymore, i.e. standalone application capability is lost. More importantly, the owner of the referencing model is not in full control because the model uses data that resides elsewhere and which can be modified independently by another party.

1.3.2 Information sharing by value

The sharing of information by value means that the information contents are duplicated inside the participating software applications. Referring to the beam example above, the length of the beam is duplicated in both the architectural and the structural models. The consequence is that an inconsistency arises when the length of the architectural beam changes after the structural beam was created.

The advantages of sharing information by value are:

- The various models can function independently, i.e. standalone application capability is maintained.
- The owner of a model remains in full control of his/her model. What happens in other models does not "automatically" change the local model.
- The owner of a model is free to interpret the incoming information and adapt it for the purposes of his/her model. For example, the structural engineer may assign a different length value to the beam to more accurately represent its support conditions.

1.3.3 Inter-disciplinary information sharing

Information sharing by reference is an attractive option due to its consistency advantages. However, the loss of standalone capability and the problems with ownership and responsibility weighs heavily against it. These disadvantages can be minimized by providing an overall control mechanism which manages and resolves the problems. However, such a mechanism would require the support and cooperation of all the role players and specifically the support of their software vendors since changes to the software would be unavoidable. The diversity of the construction industry and the large number of its software vendors make this approach very difficult. Until a solution can be devised that is accepted by the building industry role players and their software vendors it has to be accepted that software solutions will be developed independently and that sharing of information will be by value. Consequently the importance of supporting consistency in software applications that are linked using information sharing by value cannot be underestimated.

Chapter 2

Research Focus

The Architecture-Engineering-Construction (AEC) industry is an intensive user of information and significant savings can potentially be made by linking its various software applications. As a result extensive research has been directed at providing software that supports cooperation. An example is the recently completed Priority Program of the German Research Foundation "Network-based Co-operative Planning Processes In Structural Engineering" (DFG SPP 1103) [DFG-SPP 2007].

One of the results of SPP 1103 is an integrative process model described in [Rüppel 2007; Rüppel and Lange 2006]. This model is central to the management of engineering processes in terms of cooperation support and comprises the four layers shown in figure 2.1.

Communication layer: This layer models the dynamic interaction flow of information between the different participants. It provides direct access to information, based on modern communication technologies like mobile software agents.

Organisation layer: This layer models the planners and organisations involved in the process, controlling models and decision making.

Coordination layer: This layer represents the process and workflow models of the engineering process.

Resource layer: This layer contains the actual software models, i.e. their states and the rules and methods needed to process the model information.

The realisation of the integrative process model is an enormous task which requires partial solutions from a large number of researchers. It also provides a way of classifying a specific contribution, specifically in which of the layers the contribution falls.

The broad focus of this thesis is rooted in the resource layer since it deals with actual software models. More specifically it deals with the dependencies between the states of different models. Models and dependencies are defined

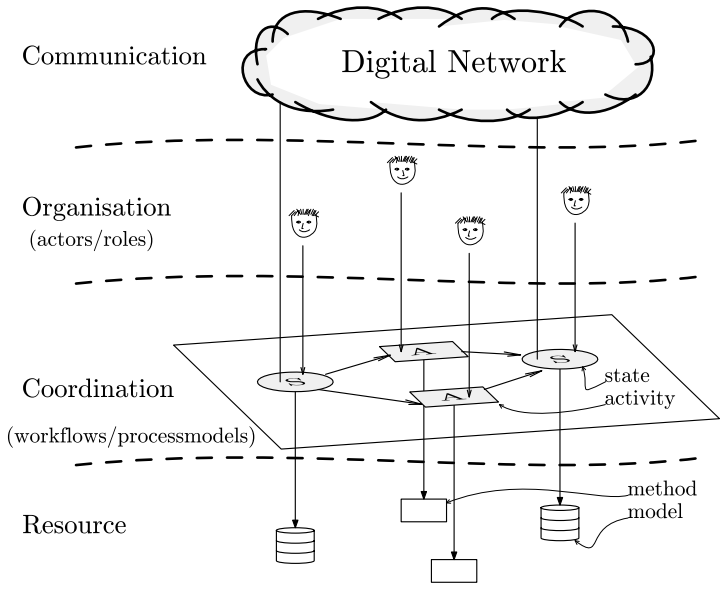


Fig. 2.1: Integrative process model

in section 2.1 below. The broad focus is to investigate what can be done to support consistency when information is shared between different models, with the emphasis on how to support the updating process in a dependent model if changes occur to upstream information.

Given the large number and diversity of software that is used in the AEC industry, the focus is narrowed by selecting two models of great importance in the field of structural engineering, namely models from the Computer Aided Design (CAD) domain and the Finite Element Method (FEM) domain. FEM models strongly depend on CAD models and in current practice a significant amount of effort is spent transferring information from CAD to FEM models. Consequently research focussed on alleviating problems associated with this interface is important. It is expected that proposed solution techniques and technologies will be applicable to other specialized domains as well.

Figure 2.2 shows the application field of the research diagrammatically with respect to CAD and FEM. It is neither focused on CAD nor on FEM. However, it is crucial to understand both environments, particularly the similarities and the differences between them in order to devise solution proposals.

The specific aim of the research is to develop a binding mechanism which accounts for the dependencies that exist between geometry specified in a CAD model and the geometry used in a finite element model employed to calculate the structural behaviour of the entity under consideration. The

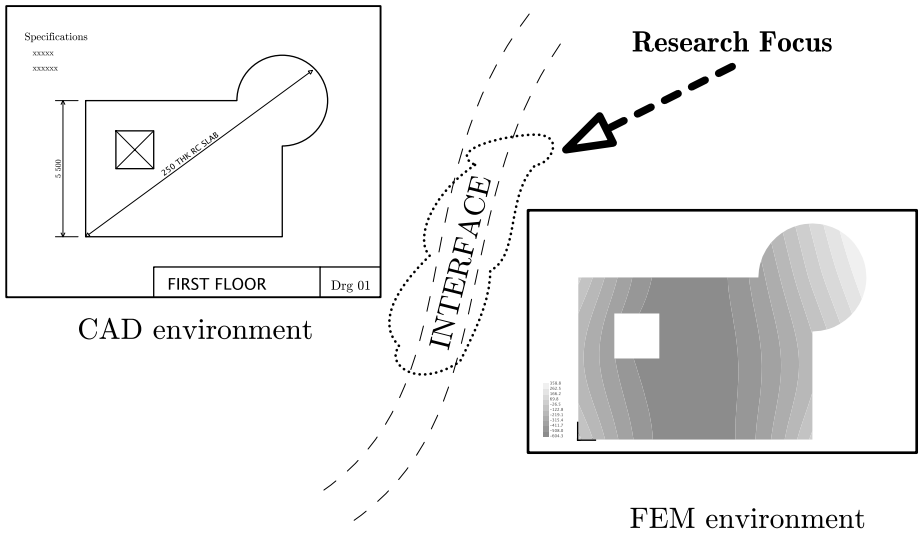


Fig. 2.2: Research focus

purpose of the binding mechanism is to assist an end-user, in this case a structural engineer, in keeping the numerical analysis consistent if changes are made to the geometry contained inside the CAD model.

The study uses an existing CAD platform, CADEMIA[1] and an Object Oriented Finite Element Framework[2] to verify the proposed solution in a pilot application.

2.1 Models and dependencies

2.1.1 Models

The results of AEC activities are elements of the built environment which are generically called products. A product model is a structured set of information objects needed to describe the product. The product model comprises the various software models of the product, where each software model is a set of information objects and the relations which serve to represent a specific view of the product. The information objects are instances of classes which describe the attributes and methods of the objects.

The following models are frequently referred to in the dissertation:

CAD model: Computer Aided Design models form the basis of every building project. It describes the building elements and the topology in the set of elements. The term CAD refers equally to a 2D drawing, a 3D object model or a 4D construction model and all are called CAD models.

[1] developed at the Bauhaus-Universität Weimar, Germany
[2] developed at the University of Stellenbosch, South Africa

A CAD model may contain information regarding client specifications, architectural layout and finishes and other non-geometric information.

Geometry model: This model is a subset of the CAD model, containing only the geometric information of the building. This model is introduced to reference the geometric part of the CAD model and does not exist on its own. It is convenient to refer to the geometry model rather than to the geometry defined inside the CAD model.

FEM model: Finite Element models contain the finite element components required to perform a structural analysis of the built instance. The Finite Element mesh is defined inside this model by the definition of elements and nodes. The geometry of the Finite Element mesh is based on the geometric information of the building. The boundary conditions, material properties and loads are part of this model. The components are used to assemble the system stiffness matrix , $[K_s]$ and the load vector(s) $\{Q\}$, from which the structural behaviour is solved. A structural engineer is responsible for this model.

2.1.2 Dependencies

In engineering applications two types of dependencies exist [Bilchuk 2005; Hanff 2003], namely structural dependencies and functional dependencies. Both of these types can by determined by an examination of the class structures.

Structural dependencies: If an object, objA, has a reference on another object, objB, then objA is structurally dependent on objB.

Functional dependencies: [Perevalova and Pahl 2004] If an object, objA, is an input value; and an object, objB, is an output value of an algorithm Z; then objB is functionally dependent on objA.

Semantic dependencies

For the purpose of the dissertation it is necessary to define another type of dependency which is denoted as a semantic dependency. Semantic dependencies are user defined dependencies between objects that otherwise have nothing to do with each other. For example, if a structural engineer interprets a line inside a CAD model instance as a beam and uses it to define the geometry of a beam inside a FEM model instance, the beam is semantically dependent on the line.

Class definitions of objects cannot support the definition of semantic dependencies due the fact that the class definitions are static and the semantic dependencies are dynamically assigned by users or algorithms during the execution of applications.

2.2 Motivation for the research

Currently some commercial software applications support users working in an integrated environment in the exchange of information between different models. However, this is limited to the suite of models provided by the software vendor and consequently it forces all the parties involved in a project to use the same software. Since a significant amount of time is usually required to master a specific software package such a solution would exclude potential participants who happen to be unfamiliar with the chosen software suite. The software-suites are usually expensive as well.

In contrast, the research described here investigates ways of using standard software applications together with a binding mechanism to support consistency between models of different domains. This prevents the exclusion of participants and allows them to use software packages with which they are familiar, without loosing the ability to keep their models consistent with other project partners. This is of particular importance to specialists using problem specific applications which may not be included in expensive *do-it-all* suites.

The research described here also applies to keeping components inside single models consistent with one another. Consider, for example, a dimension that is linked to the component it dimensions. When the component changes the linked dimension needs to be updated.

2.2.1 Re-use of information

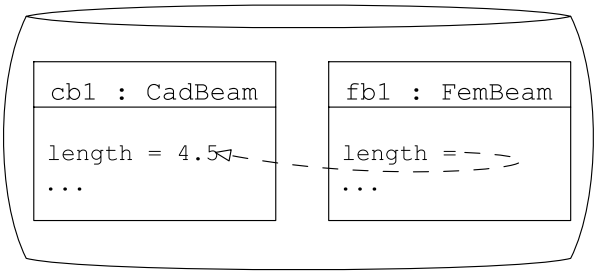
The re-use of existing information offers the potential of significantly reducing the amount of effort required to specify a dependent model, as well as eliminating the errors associated with manual specification. Only information that is not available already in digital form should be specified by the creator of a new model. In terms of this dissertation the geometric information of a CAD model instance serves as input to a geometry-dependent FEM model instance.

Consistency

The term consistency means that the state of dependent data is compatible with the state of the source data on which it depends.

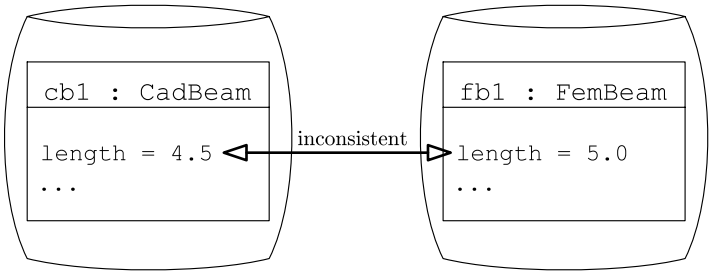
Example: Two `Beam` instances exist. The first beam instance resides in a CAD model, contains architectural information and is owned by an architect. The second beam resides in a FEM model, is used to model structural behaviour and is owned by a structural engineer. We assume that the only common information between the two beams is their length and that the structural beam is created after the architectural beam. The architect now changes the length of his beam from 5.0 to 4.5 meters.

Information exchange by reference It is clear that if the structural beam does not store the length of the architectural beam, but rather a reference to the architectural beam, consistency problems cannot occur. Each time the structural beam needs its length value, it simply obtains the length from the architectural beam. This pattern works well inside a single model. In the case where the beams reside in different models, however, the model containing the structural beam cannot function independently from the model that contains the architectural beam.



Common Model

Information exchange by value It is possible for the structural beam to exist inside a model that does not contain the architectural beam if the structural beam contains a copy of the length of the architectural beam. However, a consistency problem arises between the two length values if the length of the architectural beam changes. Information is now shared by value between the two beams. The advantage of this approach is that the models remain independent from one another and no additional complexity is introduced to try and combine the models into one *super* model. Another advantage is that the fundamental property of ownership is clearly defined using information exchange by value, because only the structural engineer can modify the length of the structural beam and vice versa.



CAD Model FEM Model

Research focus in terms of beam example

This dissertation focusses on linking the two separate domain models, i.e. the CAD model and the FEM model, for the purpose of using the existing geometry in the CAD model, namely the length of the beam, in the definition of the FEM model. In order to allow the models to remain independent and to avoid conflict of ownership, the exchange of information has to be by value. Since this may cause inconsistencies in the states of the two models, the focus is also on supporting the structural engineer in updating his model if the length of the beam in the architectural model changes.

2.3 Criteria for success

The success of proposed techniques for supporting consistency in linked CAD and FEM applications crucially depends on two aspects, namely:

- do they account for the way applications are structured and used in the AEC industry, and
- do they account for the roles and responsibilities of the persons that use them?

These two aspects lead to the following criteria:

Domain specific: Models of the AEC industry have a domain specific structure and they are used by experts in the specific domain. This situation should not be altered, i.e. models should only include information and relations that are relevant to the particular view of the product which they represent, and they should be owned and managed by persons who are competent in the specific domain. This implies that a CAD model, for example, should not be extended to contain information that is specific to FEM models since this could lead to situations where people make decisions about matters in which they have no competency. For example, if finite element boundary conditions have to be included in the CAD model a situation may arise where an architectural draftsman makes decisions about these, while it should only be done by the structural engineer.

Standalone capability: The models should remain independent of one another. Each model should contain all the domain specific information it needs and should not reference information in other models. This provides clarity about the role of the person who owns a model, and makes that person clearly responsible for the model. Furthermore it avoids the situation where a model cannot function on its own, which may cause delays and results in claims.

Apart from the main criteria listed above, the research should also produce the following:

Application independence: The proposed solutions should be independent of specific software applications. As part of the research, specific applications will be used to derive and test techniques, but the eventual proposals should be applicable to a broad class of applications.

Application specification: The properties of an application which influence its suitability for linking and updating should be listed. This serves not only as guidelines to users of software, but more importantly to software developers.

Working pilot application: Proof of the proposed solutions should be provided by developing a functional pilot application which clarifies implementation aspects of the solutions.

2.4 Structure of the dissertation

Before proceeding with the development of a solution, a brief overview of the state of the art in linking CAD and FEM models is presented in chapter 3. In order to understand the common ground between the CAD domain and the FEM domain both are analysed in chapter 4. A new model, denoted as the *Structural Engineering Model* (SEM), is introduced with a clear motivation of its necessity and the advantages it brings to consistency support and the linking of CAD and FEM models. In chapter 5 the solution approach is presented in general, abstract form. The fundamentals of the binding mechanism are described in chapter 6, dealing with important issues like change detection, update sequence determination and the functioning of the update mechanism. Chapter 7 describes the proposed solutions from the point of view of the person using its application, i.e. how a user is affected when linking and updating of CAD and FEM models are supported. The implementation details of the pilot application are described in chapter 8, followed by conclusions and proposals for further development in chapter 9.

Chapter 3

State-of-the-art

The building industry and its software providers have been struggling with the consistency of its information flows for a number of years and a large amount of research has been done. It is safe to say that according to current state of practice there is no major construction project where the complete set of information produced to plan and execute is fully consistent. There is very little formal support for ensuring consistency of information.

Information exchange formats and standardisation formats have been developed, of which an overview is presented below. Various solution-approaches to supporting continuous information flow have been investigated. In the area of linking geometric and numeric information, i.e. the focus area of this dissertation, two lines of thinking arose. The first approach is to have separate models which are derived from one another, and the second is to have a single model that contains both the geometric and the structural analysis information. Both approaches have advantages and disadvantages which are briefly discussed.

3.1 Information Exchange Technologies

3.1.1 DXF and IGES

Drawing Interchange Format (DXF) was introduced in 1982 as part of the first AutoCAD [Wikipedia 2007a] release, and was intended to provide an external representation of the data contained in the AutoCAD native file format (DWG). DXF has become a *de facto* data exchange format for 2D CAD information and 3D exchange is supported. Software vendors that are completely independent from Autodesk [Eastman 1999] implement DXF import and export functionality in order to support data exchange in a neutral file format.

The Initial Graphics Exchange Specification (IGES) was drafted by Boeing and General Electric in the late 1970's. IGES is an application independent neutral file format that has the same purpose as DXF, i.e. to exchange

CAD data between various applications using a predefined file format.

DXF and IGES are both currently used by the CAD community to exchange data. While both have the ability to transfer more than just geometry, their primary task remains the transfer of geometric information. They quickly fail when more complex information structures is to be transfered with the geometry, e.g. associative dimensioning.

3.1.2 STEP and IFC

The ISO 10303 "Standard for the Exchange of Product model data" (STEP) may be considered the most significant standardization effort in the AEC industry. STEP supports the development of a Building Construction Core Model and Application Protocols for the building industry. However, the complexity of the AEC industry and the complexity and comprehensive nature of the STEP standards have been preventing their widespread commercial implementation.

The "International Alliance of Interoperability" (IAI) was founded in 1995 with the aim of defining the Industry Foundation Classes (IFC) [IAI 2007]. The IAI decided to base development work on the EXPRESS data definition language which has been developed as an ISO standard within the STEP project. A key benefit of this decision was the immediate availability of a large body of development work in basic technologies such as geometry, as well as providing access to substantial research and development effort from many leading industry centres throughout the world that was EXPRESS based. In particular, the work on the development of the Building Construction Core Model within STEP, which had synthesized many of the results of the EU funded ATLAS and COMBI [Liebich and Wix 2000] projects, effectively moved to the IAI.

Real life entity	**Java** vs	**IFC**
abstracted to	java class	IFC class
described in	java source file	EXPRESS
graphical representation	UML	EXPRESS-G
persistent storage	serialization (binary file)	STEP file (ASCII file)
in runtime	java object	3rd party impl. required

Fig. 3.1: Mapping building components to Java and IFC compared

The IFC model [IFC 2007] is a semantic object model for the building

industry and allows the transfer of semantically rich building information. It only describes the attributes/properties of building entities, no methods are specified or provided by the IFC. Instead, all functionality to operate on IFC models must be implemented by software vendors and the operations are not formalized. In figure 3.1 a comparison is made between IFC modelling and object modelling using Java [Java 2007].

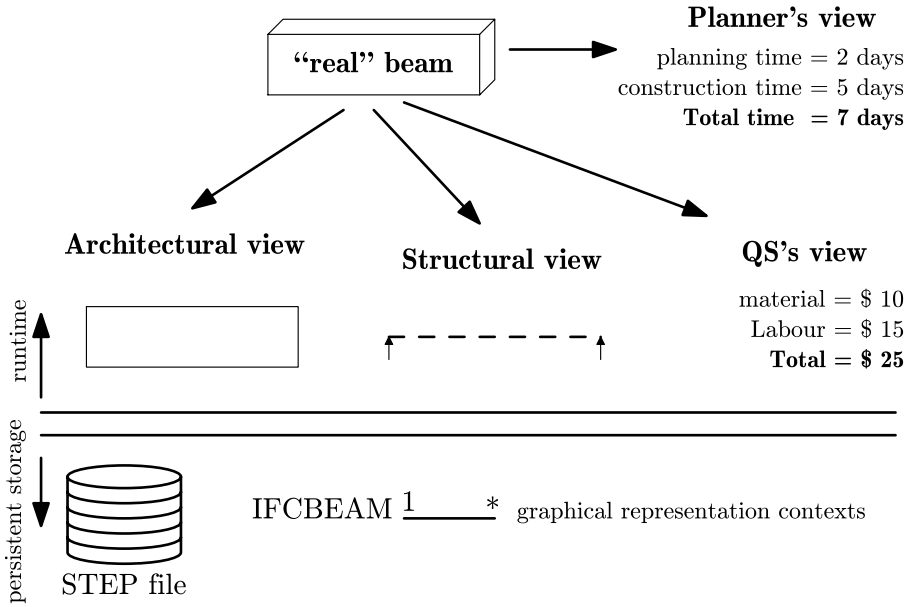


Fig. 3.2: IFC building information model

An IFC building information model with a number of different views of a beam is shown in figure 3.2. In this model, geometric information that pertains to the architectural view is stored with the view and similarly the structural view contains geometric information about the structural beam. No garuantee is provided that these two sets of information are consistent, it is left to the software vendors to provide support of consistency.

3.1.3 XML

The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages.[1]. It is classified as an extensible language because it allows its users to define their own markup tags. Its

[1]A markup language combines text and information about the text. The additional information, for example about the text's structure or presentation, is expressed using markup, which is intermingled with the primary text. The best-known markup language in modern use is HTML (HyperText Markup Language), one of the foundations of the World Wide Web.

primary purpose is to facilitate the sharing of data across different information systems, particularly via the Internet [Bray et al 2006].

XML schemas used in the construction industry are aecXML and ifcXML [IAI 2007]. Detailed describtion of these and other XML schemas falls outside the scope of this dissertation.

3.1.4 Software Agents

The term *agent* [Wikipedia 2007b] is used to describe a software abstraction, similar to Object-Oriented Programing terms such as methods, functions, and objects. The concept of an agent provides a convenient and powerful way to describe a complex software entity that is capable of acting with a certain degree of autonomy in order to accomplish tasks on behalf of its user. Unlike objects, which are defined in terms of methods and attributes, an agent is defined in terms of its behavior.

Various authors have proposed different definitions of agents, these commonly include concepts such as

persistence: code is not executed on demand but runs continuously and decides autonomous i.e. by some internal mechanism, when it should perform some activity.

autonomy: agents have capabilities of task selection, prioritization, goal-directed behaviour, decision-making without human intervention.

social ability: agents are able to engage other components through some sort of communication and coordination, they may collaborate on a task.

reactivity: agents perceive the context in which they operate and react to it appropriately.

The use of software wrapper- and migrating agents can support the exchange of semantic rich information, including attributes and methods, between different software applications and platforms [Alda et al 2004].

3.2 Current research themes

This section gives a short overview of ongoing research in the field of civil engineering informatics.

3.2.1 Distributed Product Models

This field can be divided into two categories, namely distributed models and knowledge models.

Distributed models

Modeling of structured set of object versions: [Beucke et al 2007; Firmenich 2002] This approach uses object versions and bindings between these object versions to model a product model as a graph. Operations are defined that operate on this graph to ensure consistency. Thus, a product model is versioned at object level and not at file level as before. It supports reciprocal work i.e. more than one person working simultaneously on the same set of objects.

A system architecture for distributed models and applications was developed by [Beer 2006].

Operative Modeling: Information lost is a well known phenomenon when evaluated models are exchanged. A different approach, that is to exchange operative models is investigated [Koch and Firmenich 2006]. This approach exchanges the operations applied to the model rather than the model itself. The operations that are performed are journalled, i.e. recorded in such a system, and could be executed again in another system.

Analog to this, a 3D model instance that could either be represented by a BRep[2] (evaluated model) of CSG[3] (operative model).

Volume based Modeling: [Rank et al 2007; Romberg et al 2004] This approach suggests that a building instance should be modelled completely as a 3D volumetric model, without the reduction of dimension. This is made possible by the ever increasing availability of computational power. This approach assists the integration of sub models with the central geometric model.

Knowledge models

Knowledge model instances are useful to add knowledge to the planning process during runtime. Without knowledge models, all knowledge must be hard-coded by the programmers beforehand. Some examples where knowledge models support the planning process are:

Conceptual design support: Knowledge models are used to map the semantics of the building instance to the computer during early design phase [Kraft and Nagl 2007; Kraft and Retkowitz 2006]. Thus, the conceptual design also captures the design intent. This knowledge is then used during the remainder of the project to verify the design as the development of the product continues.

[2] boundary representation
[3] constructive solid geometry

Verification against building standards: Another example of the utilization of knowledge models is the verification of a design against building standards and design formulas located in a knowledge model system [Schnellenbach-Held et al 2004]. These knowledge model instances are then used to analyse the consistency of product models with regards to expert engineering know-how and building standards.

3.2.2 Process Modelling

A process model is used to model the processes associated with the design and execution of a project. A process model comprises of actors, activities and states.

In [Tauscher et al 2007] different execution alternatives for a specific project are modelled in one model. This approach allows the generation of a construction schedule that contains alternative processes to achieve the same end goal based on a set of independent construction tasks. Product and process models are loosly coupled with the assistance of hierarchical graphs [König 2004].

An example where Petri Nets are used to model workflows can be found in [Katzenbach et al 2006]. Here the cooperation model is divided into local process domains, information packages and a global process management system. Information exchange between different planners is organized in information packages which consist of task specific information e.g. geometrical information and some meta-information providing the necessary information for the control of the processes. Based on the underlying process model, the process simulation and by evaluating the metainformation the global process management system dynamically activates communication channels between different engineering organizations.

During the execution of planning processes some unexpected problems might occur. This leads to the introduction of additional activities in the relational process model that describes the process to resolve the problem [Klinger et al 2006]. This problem is addressed by the creation of a second step of an activity in order to avoid cycles and is formalized to ensure that the process model remains consistent after the modifications.

3.2.3 Distributed simulation

In this context, the term *simulation* is defined as computer based modelling of engineering problems in the structural engineering domain. Three categories are identified in terms of the DFG Priority Programme 1103.

Numerical

This branch of distributed simulation contains the numerical approximation of differential equations that describe a continuum for example the Finite

Element Method, as well as other computer supported methods like computational geometry.

Event based

Event based simulation focuses on the modelling of discrete components in a complete system. The interaction between different components are modelled and individual events control the system as a whole, for example agent systems or graphical user interfaces.

Analytical

Problem definitions are solved symbolically with the use of computer algebraic systems for example MATLAB, Mathematica and Maple.

3.2.4 Agent systems

Agent systems are well suited for the modelling of complex systems in a dynamic environment. In [DFG-SPP 2007] five agent-based sub models are defined to support the structural design process. These are agent-based cooperation-, product-, process-, software integration- and expert knowledge models. These five sub models fits well into the different levels of integrative cooperation process model shown in figure 2.1.

Another example of an agent system that was developed to connect different models for cooperative building design [Rüppel and Lange 2006] is a network-based fire engineering support system.

3.3 Information flow inside a project

When computers first started to be used in structural engineering, CAD and FEM applications were developed as standalone applications with their individual data structures and file formats. At that time the exchange of data between applications, i.e. integration of different applications, was not as important as the development of the applications themselves, resulting in application specific algorithms and data structures. With the dawn of the Internet era, however, the focus of application development moved towards using network technologies to support the exchange of information between different applications that were developed for specific, standalone tasks. The development of a link between CAD and FEM applications followed the derived model approach and the single model approach.

3.3.1 Derived model approach / Data exchange

The first approach in moving data from one system to another was the development of standardized interfaces between different systems to create a

common ground. The applications were extended to represent their internal data in a prescribed format, e.g. DXF in the case of CAD information. (See section 3.1)

In this environment, information migrates from task to task throughout the project development. Each task reads information from a source, processes the information and eventually stores the results in another file which is read in turn by the next task, or it creates output on another device e.g. a printer.

Figure 3.3 presents an example of re-using CAD defined geometry inside a FEM application. The FEM application reads the geometry from a file that was created by the CAD application, and that serves as the starting point for the creation of the FEM model.

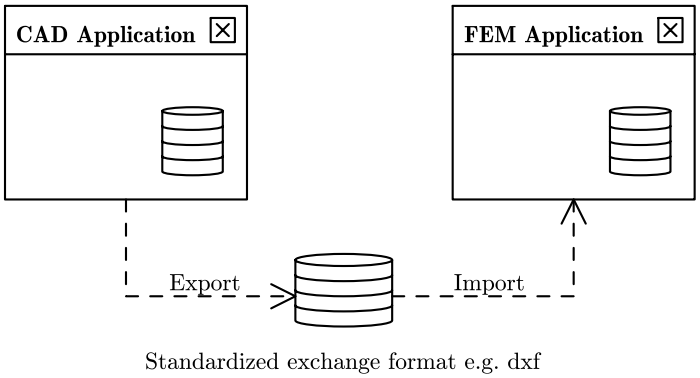


Fig. 3.3: Exchanging information via exporting and importing

3.3.2 Single model approach

The main aim of the Building Information Model (BIM) is to centralize all information with regards to a project in a single model that comprises different submodels, see figure 3.4. This information could span the complete life-cycle of the building under consideration.

Each participant in the project reads required information from the BIM, performs some tasks, and stores the results back into the BIM where it is made available for other parties to use as input for their specific design tasks.

The IFC model server is a good example of development in the area of supporting the construction industry with BIMs. It uses the Industry Foundation Classes as format for the exchange of information. The server implementation is responisible for the management of IFC data, i.e. the server provides partial product models on demand to users to work on and merge the modified partial models back into the BIM once the work on a specific partial model has been completed.

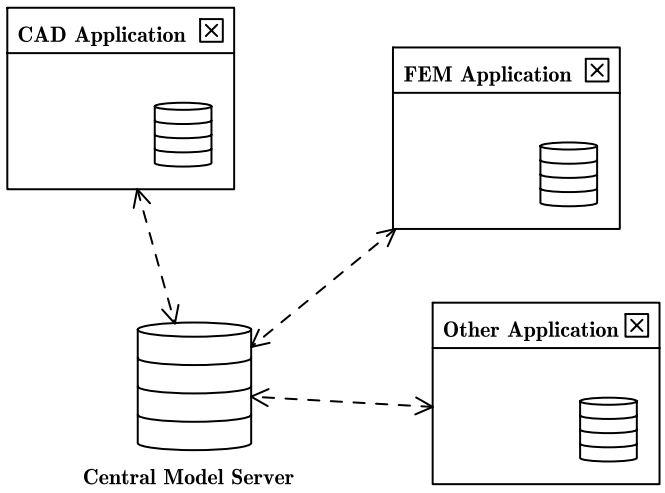


Fig. 3.4: Building information model

An example of a single application that contains information that belongs to more than one domain is Autodesk's *Revit Structures* [Autodesk 2007]. It uses the BIM approach to integrate the geometric and the numeric model into a single model, in this case a .dwg file. A case study that discusses the use of this product is described in [Senescu et al 2006].

Revit Structures uses objects that contain structural semantics to model the building instance, e.g. columns, walls and slabs. In addition to this it allows the user to add components from the structural analysis domain like frame elements. These components are then accessed via an API from the external analysis application. The user interface allows these components to be specified simultaneously.

This case study uses an external finite element application, *ETABS* [ETABS 2007] to perform the actual numerical analysis outside the model. *ETABS* imports the structural definition via an API supplied by *Revit Structures*, and after the import additional work needs to be done inside the *ETABS* environment to finalize the numerical model of the structure. Thus, *Revit Structures* only assists in the definition of the structural components that coincide with the geometry of the structure.

An additional point to note is that the *Revit Structures* model is file based. That means that the exchange of information can only be done at file level. Thus, when two or more people work together on the same model, each person would have to work on his own copy of the file or if the same file is used by all role players, only one could work at a time to ensure that everyone works on the same information.

Additional work done in the *ETABS* environment cannot be ported back to the *Revit Structures* model. The *Revit Structures* approach is a step to-

wards a single model that contains both the geometrical and the analysis information. At this point in time, the actual numerical analysis is performed by a standalone application outside *Revit Structures* and changes made outside *Revit Structures* cannot be ported back into *Revit Structures* itself. Although *Revit Structures* is a good productivity enhancement tool, it does not attempt to solve the problem of information consistency fundamentally.

3.4 Supporting consistency

Both approaches explained above work well in an environment that is free of changes to the geometry. However, once changes occur to the geometry the numerical model has to be updated to account for the changes.

In standalone systems, a user needs to re-import the geometry or needs to update the geometry manually in the FEM application to ensure that the geometry on which the FEM analysis is based reflects the geometry that the CAD system dictates. This is a complicated and error prone task and sometimes it is easier to start the numerical analysis over from the beginning.

A BIM model that contains both the geometry and the numerical components also requires additional work by a user but, since both the geometric and analysis components are in the same model and the same application acts on this model, the updating of the analysis components are better supported than in the standalone case. However, inconsistencies can still easily arise.

Chapter 4

Application models for geometry and structural analysis

Applications for finite element analysis and computer aided drafting have been under development for almost five decades. As a result both application types are well developed and each offers a wide range of advanced features. Bridging the gap between CAD and FEM applications may have been hampered because of their advanced state of development in differing directions. In this chapter the focus is on identifying the essential aspects, that fall within the scope of the dissertation, of both application types. These aspects are the ones that involve geometry. Consequently the characteristics relating to geometry in both the CAD and the FEM environments should be understood before an attempt can be made to link the two. These characteristics are discussed and an argument is made for the introduction of a new model, the Structural Engineering Model (SEM), if consistency of geometric information is to be supported. However, the realization of a Structural Engineering Model and the development of a solution concept require a more detailed analysis of FEM and CAD applications. FEM applications are addressed first, with a specific focus on the visualization and graphic manipulation of FEM models. The structure of CAD applications are then analysed from the viewpoint of linking geometry-dependent models, like FEM models, into the CAD domain. A proxy-based design pattern to achieve such a link is described and illustrated with an implementation example.

4.1 Discrete and continuous geometry

From the viewpoint of the structural engineer it is useful to distinguish between two types of FEM models with regards to their geometry contents,

namely discrete models and continuous models. The discrete models refer to truss and frame structures, which, by their nature, requires limited effort in terms of finite element discretization (meshing). In many instances the physical truss or frame element is mapped to a single one-dimensional (1D) finite element, in which case a modified analysis technique is used which yields exact results of truss and beam theory in spite of the coarse meshing. It should be noted that truss and frame structures may also be analysed using e.g. 2D shell or 3D solid finite elements of elasticity theory, but the vast majority of work is done using the 1D truss and frame elements.

Continuous FEM models are used when the geometry of the problem is more general, i.e. comprising floor-slabs, walls, etc. In these cases 2D and 3D type elements are employed to interpolate the geometry and physical state. This requires considerable effort in terms of meshing of the problem domain. The results of such a model are highly dependent on the quality of the mesh and the type of element used in the analysis.

Both model types are discussed by way of examples and with reference to CAD and FEM applications.

4.1.1 Discrete models

Figure 4.1 presents a model of a truss. This view, with some additional information like dimensions and member sizes, is sufficient for a contractor to create construction drawings and to prepare for the actual manufacturing of the truss. From an application point of view there exists a model which contains information that could either be displayed on a screen or printed out.

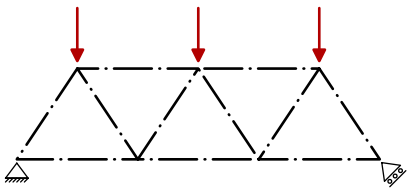


Fig. 4.1: Truss example

CAD model

In the 2D CAD environment, the truss is represented by a number of lines that indicate the members, some text that describe the member sizes and dimensions that provide the member lengths.

In this environment, two lines are connected if they have the same end point coordinates. The semantics are interpreted by persons viewing the

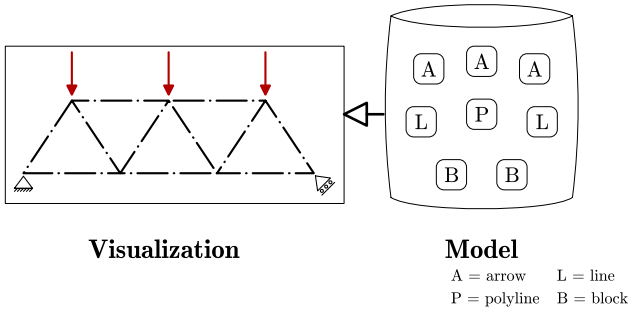


Fig. 4.2: CAD application

drawing, and are based on standards that developed over time. For example, the diagonal members may either be represented by one polyline that touches the top and bottom lines or by 6 straight-line elements. While this choice resides with the creator of the model, the semantics of the truss are unambiguously communicated to the manufacturer independently of the actual CAD objects used to create the model. Figure 4.2 shows the graphical view with the underlying CAD objects that model the truss. It is important to note that no dependencies exist between the objects of the example model. In general, however, there may be dependencies between components of a CAD model. For example, in the case where associative dimensioning is implemented, a dimension depends on the component it is associated with, thereby making sure that a consistent view is maintained.

FEM model

In the numerical analysis environment, the geometry of the truss is completely described by the coordinates of the truss' nodes. The topology of the individual truss elements and the geometry in terms of nodes describe the layout of the truss elements.

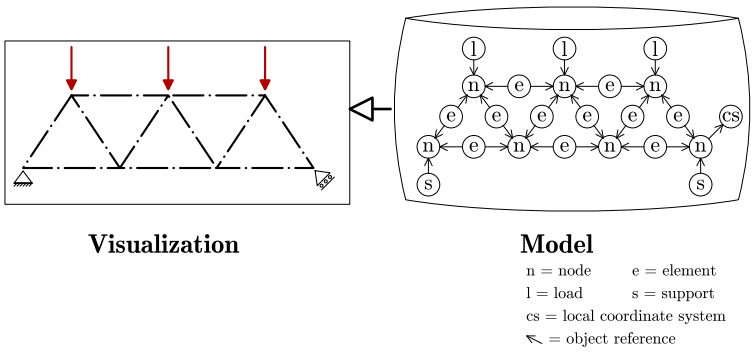


Fig. 4.3: FEM application

If the underlying application is a FEM application, there would be exactly 11 truss elements connected to 7 nodes. This model also contains 2 supports and 1 local coordinate system which accommodates the skew support on the right-hand side of the truss. In the numerical analysis domain, elements are connected when they share an end node and unconnected when they have different end nodes, even if these happen to have the same coordinates. An applied load only "knows" the node on which it acts in addition to its own attributes, e.g. its direction and intensity. Thus a load is applied via topology and not geometry. The same applies to support conditions and local coordinate systems. Figure 4.3 shows the truss as well as the underlying FEM model. The dependencies between the objects in the model are represented by arrows. A truss element depends on two nodes, while loads, supports and local coordinate systems each depends on one node.

Conclusions

The following conclusions can be drawn from the preceding truss example.

CAD model There is often no explicit modelling of topology in the geometry of the CAD model. The creator of the CAD model could choose different CAD objects to achieve the same end result. Whether these objects are primitive CAD components or more semantic components makes no difference to the problems addressed in this dissertation.

FEM model The topology of the components together with the geometry of the nodes define the truss. The components of the model depend on one another for their own existence, e.g. an element can not exist without it's nodes.

Deriving FEM from CAD If the creator of the CAD model complied with some simple guidelines, it would have been possible to create a part of the FEM model automatically. If the CAD model contained exactly 11 lines that represented the 11 truss members and the endpoints of these lines shared the same coordinates where they meet, it is possible to convert the line endpoints into FEM nodes and the lines into FEM elements. Since the truss is a discrete model, no additional nodes are required to describe the displacement field of the truss. Further input on the FEM side includes the definition of the cross section properties of the members, the material properties, the load intensities, the local coordinate system and the support conditions.

Separation between CAD and FEM If the CAD application supports the specification of the additional information as part of the CAD model, it

is no longer a 2D CAD application, but a FEM application with a drafting front-end. Furthermore the role of the model creator changes from draftsman to structural engineer. In practice a FEM application could import the geometry of the 2D CAD model as starting point for the FEM model. If the geometry in the CAD model complies to the export/import specifications, it can be transformed directly into FEM nodes and elements. The FEM application is then used to create the additional FEM components needed to perform the structural analysis.

Consistency problems Assuming that a structural engineer creates a FEM model by importing an existing CAD model and the CAD model changes, the structural engineer needs to update the analysis model to maintain consistency between the CAD representation and the numerical analysis of the truss. Depending on the flexibility of the FEM application it may require significant effort to update the model without disposing of work done on the FEM model since the original import.

4.1.2 Continuous models

In contrast to discrete models, continuous models require significant finite element meshing effort on the FEM application side. It is still possible to import geometry from a CAD model, but this geometry cannot be used to create a FEM model without the addition of more finite element nodes and elements.

Considering the example of a floor slab as shown in figure 4.4, there are several different ways to model the slab in CAD that would lead to the same drawing in the end.

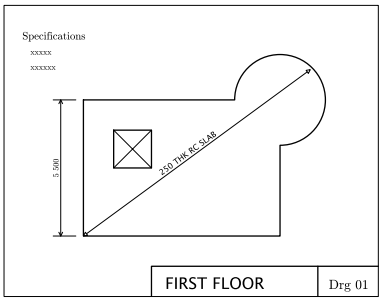


Fig. 4.4: CAD representation of a slab

The creator could either use primitive geometry like lines, arcs and polylines (2D CAD environment) or solids (3D CAD environment). The actual slab is abstractly described by some geometry and other non-geometric components, e.g. text that indicates the thickness and dimensions. Additional information required to construct the slab is the reinforcement and material

specifications, which becomes available after the analysis and reinforcement design of the slab.

In order to perform an analysis and design, information regarding the slab is transferred to the FEM application. The first task for the structural engineer would be to create a mesh that fits the geometry *and* the intent of the FEM analysis. If the FEM analysis is performed to determine the reaction forces supporting the slab, the mesh could be rather coarse compared to a mesh that should give an accurate approximation of the shear stress distribution at the supports. Assuming that the CAD model does not contain any structural information that can be directly ported to the FEM model, the creator of the FEM model has to perform the actions required to create the appropriate mesh and to specify the remaining components of the FEM model. The definition of boundary conditions, the loading on the slab as well as the material properties are all tasks that should be executed by an experienced structural engineer.

Figure 4.5 shows the result of a meshing procedure performed on the geometry of the slab to obtain a mesh that can be used for the analysis of the slab. This mesh is refined near columns that support the slab to give a closer approximation of the displacement field in the area of sharp curvature gradients around the columns.

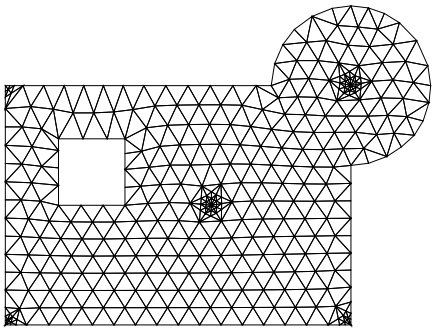


Fig. 4.5: Meshed slab

4.1.3 Connecting FEM to CAD models

The main problem in connecting CAD and FEM models is that geometry plus annotations, e.g. the shape and thickness of a floor slab, are the only common denominators between the two. In discrete models a mapping of CAD entities to FEM entities may exist if and only if the creator of the CAD model adheres to a set of guidelines while creating the CAD model, as described in section 4.1.1. In general this is not the case. For continuous models a mapping is not possible, since a finite element mesh is required

which not only depends on the geometry and annotations, but also on the type of element used, the loading pattern, support conditions, etc.

Binding CAD and FEM information

The goal of linking and binding CAD and FEM information would be to support the transfer of information and updating of the FEM model if the CAD model changes. For example, if the radius and center point of the arc changes in the preceding slab example (i.e. a CAD change occurs), the FEM model should be updated to reflect these changes. Otherwise, the FEM model is outdated and inconsistent with the geometry that defines the actual slab. As explained in the preceding sections however, all the components that define a finite element model, e.g. nodes and elements, do not and should not exist in the CAD model, and cannot be derived from the CAD model directly. The CAD geometry has to be combined with structural engineering input to create the finite element model, as shown in figure 4.6. A drawing represents an interface which is *interpreted* by a structural engineer when the entity under consideration is designed. CAD information is effective in communicating a layout view of an entity, however re-using the information in a design application requires more.

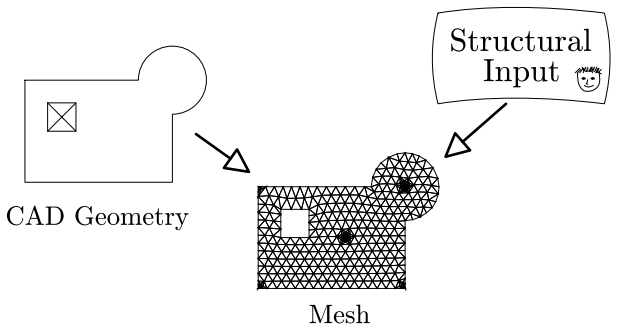


Fig. 4.6: Creating a Finite Element Model

The consequence of this is that a direct CAD-FEM binding relation cannot be devised. To overcome this problem, a new model is introduced, namely the Structural Engineering Model (SEM), described below.

4.2 Structural Engineering Model (SEM)

The importance of keeping models consistent is apparent. However, it is clear that it is not viable to bind FEM models directly to CAD models due to the lack of finite element specific information inside the CAD model and the burden it would place on the person responsible for the CAD model.

These problems can be dealt with by the introduction of another model, denoted as the Structural Engineering Model (SEM). The SEM exists on its own, and provides the middle ground between the CAD model on the one side and the FEM model on the other. The core task of the SEM is to encapsulate the CAD geometry, to support the merging of structural semantics with the geometry, and to provide front-end functionality for the FEM model. The person responsible for this model should be a structural engineer. Figure 4.7 shows the information sources of the SEM.

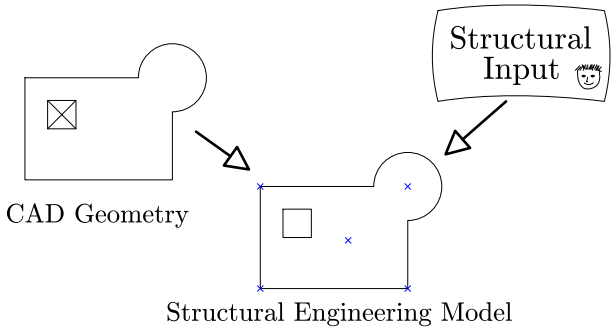


Fig. 4.7: Structural Engineering Model

The engineer uses the SEM to define a FEM model in an abstract way. Thus, the SEM contains structural components like slabs, beams, columns, loads, supports, etc. Where applicable, these components are related to geometry objects imported from the CAD model, e.g. 4 lines may be identified as the edges of a slab. The engineer also specifies other parameters that comply with the requirements of the finite element analysis, after which a FEM model is derived from the SEM.

As described above, the Structural Engineering Model does not differ from state-of-the-art pre-processing models for finite element applications. However, the SEM provides the opportunity to solve crucial aspects of the consistency problem set out as the particular focus of the research described in this dissertation. Furthermore, the finite element pre-processors are developed and maintained at great cost, which eventually spirals down to the users, namely the engineering offices. In addition these offices also have to acquire costly CAD software. An approach where CAD functionality can be used to substitute the costly finite element pre-processors also offers the potential of direct cost saving. In addressing these issues, the SEM forms the backbone of the concept for the integration of CAD and FEM models which is proposed in the next chapter.

Before proceeding to the integration concept and details of the SEM, some additional background regarding FEM and CAD models is required. The SEM has to bind geometric information of the CAD model to analysis entities of the FEM model. This requires an understanding of the stucture

of FEM applications, especially with respect to finite element pre-processing and graphical manipulation. In the same vein the structure of CAD applications has to be analysed, and a model to tie entities from both application spheres has to be devised. These issues are discussed below.

4.3 Finite Element Applications

The first finite element applications were console-based programs that read input, in the beginning from punch cards, later from input files. It then assembled and solved the system equations and computed the nodal displacement and element stresses and strains. The results were text based output into files or on a printer. The research and development focus was largely on effective assembly, storage and retrieval and solution of the system equations in the absence of sufficient computer memory and processing power. The following three steps that comprised a finite element analysis were clearly visible:

- pre-processing
- processing
- post-processing

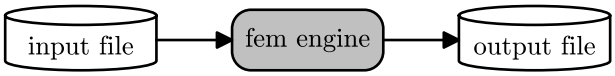


Fig. 4.8: Simple FEM Application structure

As processing speed and core memory increased, and peripheral hardware devices like high resolution displays and printers developed, the focus shifted to the pre- and post-processing steps, which quickly caught up with the processing step.

Modern finite element applications still follow this pattern, although it may not be visible to an end-user. Typical finite element applications have graphical user interfaces (GUIs) that provide end-users with powerful tools to create and modify geometry, boundary conditions and loading patterns. The analysis part of such an application is often hidden by the GUI. The transition between pre- and post-processing may not even be noted by the user. He/she would typically press the analyse button and in the case of a small model, the results would be available for evaluation before the user noticed the 'analysing model' message.

The three steps are now described in more detail.

4.3.1 Pre-processing

The pre-processing phase of a finite element analysis can be defined as the mapping of the model of the structural entity under investigation to finite

element components.

Three ways of supporting pre-processing are available:

Text input files Traditionally, this was done by the preparation of input files in a text editor. These files typically contained the nodal coordinates, element definitions, boundary conditions and loading conditions. This completely separates the pre-processing and processing phases.

GUI with tables With the development of graphical user interfaces, graphical front-ends were developed that provide the user with tables to specify the information previously specified in files, but still in text format. Typically, a picture of the actual state of the model as the user progresses is also shown. The picture, however cannot be used to alter information entered in the model. Model modification can only be done via the tables.

GUI with drafting capabilities The third approach is to support specification of the model via graphical tools, in addition to text and tables. This type of pre-processor has the look and feel of a traditional CAD application. However, instead of drawing a CAD line, a user draws, for example, a frame element.

In the second and third approaches, the finite element model may either be stored in files like the first approach, or the intermediate text file storing may be omitted and the processing step may be invoked straight away.

During pre-processing, tools like meshers, for example, are used to assist the structural engineer in defining the finite element model. Higher end of the market applications allow users to define a finite element model at a more abstract level than nodes, elements and supports, namely through the creation of macro elements, e.g. part of a slab with some meshing parameters.

4.3.2 Processing

The processing phase of a finite element analysis involves the parsing of the input file or the evaluation of the model data that was specified by tables and/or constucted graphically via a GUI. It then assembles and solves the system equations. Once the equations are solved, element results are computed, e.g. stresses and strains in the elements. The main computational effort is spent in solving the system equations. A clear separation between the pre-processor and the processor is achieved by using text input-data files. The specification of the contents of the input file enables the use of different pre-processors, as long as the defined file format can be created.

4.3.3 Post-processing

The end result of the analysis step is that the primal value at each degree of freedom is known, as well as dual values at the points where primal values

have been specified. This information is processed further to create information that is useful in understanding the structural behaviour, e.g. maximum deflections, stresses and bending moments. Alpha-numerical results are important for the design step that follows the analysis step. However, figures and diagrams are indispensable for an engineer to understand and interpret structural behaviour.

4.4 Visualizing a FEM model

The visualization of a FEM model displays the geometry and topology of the model components, as well as the results. This is useful to verify that the model data is correct before performing the analysis and to interpret the physical behaviour of the entity under investigation.

Keeping the functionality to visualize a model outside the model itself is good software development practice. This separation allows a user to use different visualization tools for different tasks, while using the same underlying FEM model. Also, changes to the visualization software do not trigger changes to a FEM framework that is stable and thoroughly tested.

The technicalities of visualizing a FEM model are described below. This is done to provide insight into the state of technology available to developers of visualization software, and to show that manipulation of a visualized model entails a major increase in complexity and programming effort. This supports the argument for using CAD applications to provide visual manipulation of engineering models like FEM. Furthermore it is clear that the requisite separation between visualization and model coordinates well with the concept of using CAD applications for the visual manipulation of FEM models.

4.4.1 Visualization using the Java2D API

The Java2D API is a powerful technology that supports the visualization of objects. A detailed description of the use of this API falls outside the scope of this work.

Creating a visualization component

The creation of a visualization component is done by extending the `JComponent` class which is part of the javax.swing package. Figure 4.9 shows the UML modeling of a FEM visualization component called the `FemPanel`. Since it extends `JComponent`, the `FemPanel` is handled in the same way as any other Swing component. The intention is that when an instance of this class is created and added to a `JFrame`, the underlying FEM model is displayed.

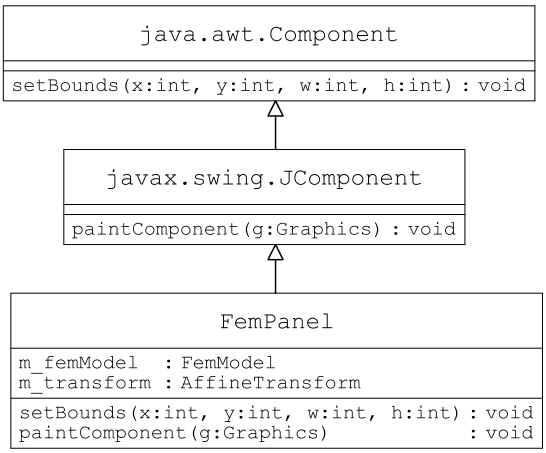


Fig. 4.9: FemPanel

Figure 4.9 shows two attributes and two methods of the `FemPanel` class. The different roles of the attributes and methods in visualizing a FEM model are described below.

m_femModel This attribute references the FEM model which is displayed by the `FemPanel` and is assigned during the construction of the latter. It allows the panel to access the finite element model's information that has to be displayed.

m_transform This attribute represents a coordinate transformation. Before a formal definition of the attribute can be given, the three different coordinate systems involved in the visualization of a FEM model must be understood:

model coordinate system (MCS): This is the coordinate system of the FEM model. The units of this coordinate system are physical length units like meters or millimeters.

panel coordinate system (PCS): This coordinate system is a device independent coordinate space of the `Graphics` instance handed to the `paintComponent(Graphics g)` method. The units of this coordinate space are points where 1 point = 1/72 inch.

device dependent coordinate system: This is the coordinate system of the device that is used to display the model. The units are specified in dots per inch (dpi) where a screen would have 96 dpi and a printer much more, e.g. 1200 dpi.

In the case where 3D models are visualized a coordinate transformation is performed based on a viewpoint and view direction. The result of this

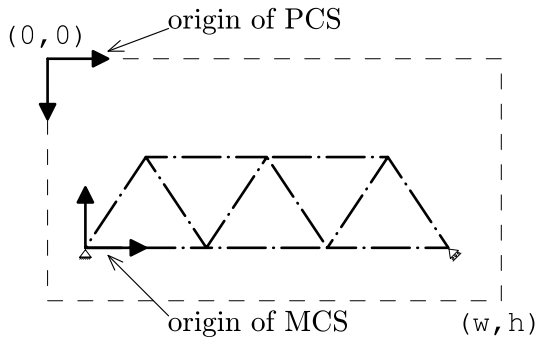


Fig. 4.10: Model and panel coordinate systems

coordinate transformation is a 2D projection of the model in the MCS. The transformation from MCS to PCS has to be taken care of by the implemetation of the `FemPanel`.

The attribute `m_transform` is of type `AffineTransform` and is used to perform the transformation from the MCS to the PCS of the `Graphics` instance. The second transformation, i.e. the transformation from the PCS (device independent) to the device dependent coordinate system, is taken care of by the `Graphics` instance handed to the `paintComponent(Graphics g)` method of the `FemPanel` instance.

Figure 4.11 shows six of the methods of the `AffineTransform` class. The first four methods are used to define a coordinate transformation between two different coordinate systems. Once the coordinate transformation is defined, the method `createTransformedShape(Shape pSrc)` transforms a given `Shape` [1] and the method `transform(Point2D ptSrc, Point2D ptDst)` transforms a given `Point2D` instance using the defined coordinate transformation.

java.awt.geom.AffineTransform	
rotate(theta:double)	: void
scale(sx:double, sy:double)	: void
shear(shx:double, shy:double)	: void
translate(tx:double, ty:double)	: void
createTransformedShape(pSrc:Shape)	: Shape
transform(src:Point2D, dst:Point2D)	: Point2D

Fig. 4.11: AffineTransform methods

Zooming and panning functionality can be included in the visualization panel by manipulating the `m_transform` instance and repainting the model.

Two methods of the `FemPanel` class shown in figure 4.9 are described below:

[1] see section Shape Interface below

setBounds(...); This method of class `java.awt.Component` is called by the `LayoutManager` of the container that contains the component. The `LayoutManager` assigns the bounding box of the component based on the surrounding user interface (UI). Before the UI is displayed the first time and each time the size of the UI is changed, this method is invoked on all the components to assign new sizes. The bounding box of the FEM model is known by the `FemPanel` and the bounding box of the component itself is passed to the component by this method. Using this information the transformation between the MCS and the PCS is calculated in this method by overriding the method in class `FemPanel`.

The implementation first calls the `super(...)` method, then instantiates an `AffineTransform` instance and defines the coordinate transformation by calling the `scale(...)` and `translate(...)` methods with the calculated values as shown in listing 4.1.

```
public void setBounds(int x, int y, int width, int height) {
  super.setBounds(x, y, width, height);
  double scale = Math.min(height/m_modelBounds.getHeight(),
    width/m_modelBounds.getWidth());
  m_transform = new AffineTransform();
  m_transform.scale(scale, -scale);

  m_transform.translate(-m_modelBounds.getX(),
    -m_modelBounds.getHeight()-m_modelBounds.getY());

  // center transform
  if((getHeight())/m_modelBounds.getHeight() <
    (getWidth())/m_modelBounds.getWidth()) { // center x
      m_transform.translate((getWidth()/scale-m_modelBounds.getWidth())/2.,0);
  }
  else { // center y
    m_transform.translate(0,
    -(getHeight()/scale-m_modelBounds.getHeight())/2.);
  }
}
```

Listing 4.1: setBounds(int x, int y, int w, int h)

paintComponent(...) The actual rendering of any component is done by this method. When the `repaint()` method of a component is invoked, the following three methods are called:

- `paintBorder(Graphics g)`: paints the border of the component
- `paintComponent(Graphics g)`: paints the component itself
- `paintChildren(Graphics g)`: paints the children of the component i.e. the components contained inside this component.

In the `FemPanel`, the `paintComponent(Graphics g)` method inherited from the `JComponent` class is overridden to perform the actual rendering of the FEM model. The first thing that this method does is to call the `paintComponent(Graphics g)` method of the super class. This ensures that the background of the component is correctly painted and the `AffineTransform` instance inside the `Graphics` instance it correctly set.

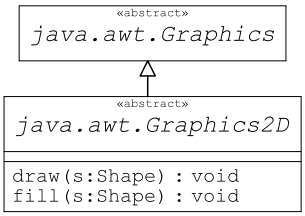


Fig. 4.12: Graphics and Graphics2D

Next, the `Graphics` instance is cast to `Graphics2D`. This makes the `draw( Shape s)` and `fill(Shape s)` methods available, as indicated in figure 4.12. The drawing environment is now set and the actual drawing of the `FeModel` can start. Shapes are rendered in the PCS and the creation and transformation of the shapes are described below.

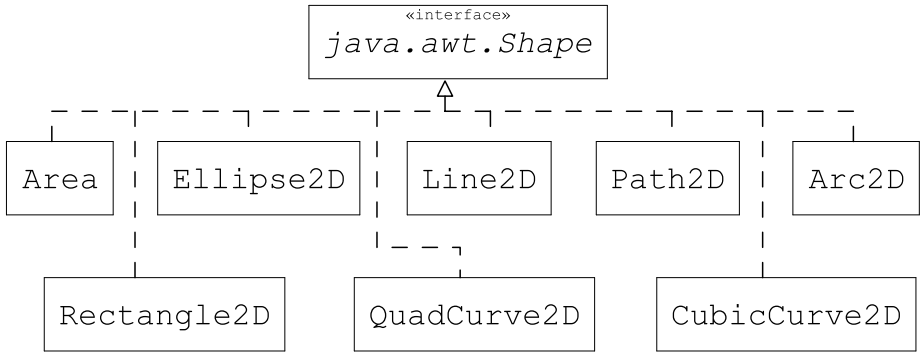


Fig. 4.13: The `Shape` interface

Shape Interface: The `Shape` interface provides definitions for objects that can be represented in geometric form. The `Shape` is described by a `PathIterator`, which can express the outline of the `Shape` as well as a rule for determining how the outline divides the 2D plane into interior and exterior points [Java 2007]. Figure 4.13 shows some of the `java.geom` classes that implement the `Shape` interface.

When the `FemPanel` is rendered, the `paintComponent(...)` method traverses the FeModel instance and maps each FeComponent to one (or more) `Shape` instances. These `Shape`s are then transformed to the PCS and displayed on the `FemPanel`.

The procedure for displaying a single component of the FeModel is given below. Refer also to listing 4.2 for the Java code:

1. get an FeComponent

2. create a `Shape` that represents the FeComponent (in the MCS)
3. transform the `Shape` to the PCS
4. draw the `Shape`

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
  Iterator iter = m_model.iterator(new TypeFilter(TrussElement.class));
  while(iter.hasNext()){
    TrussElement te = (TrussElement)iter.next();
    double x1 = te.node(0).getX();
    double x2 = te.node(1).getX();
    double y1 = te.node(0).getY();
    double y2 = te.node(1).getY();
    Shape _shapeMCS = new Line2D.Double(x1,y1,x2,y2);
    Shape _shapePCS = m_transform.createTransformedShape(_shapeMCS);
    g2d.draw(_shapePCS);
  }
}
```

Listing 4.2: paintComponent(Graphics g)

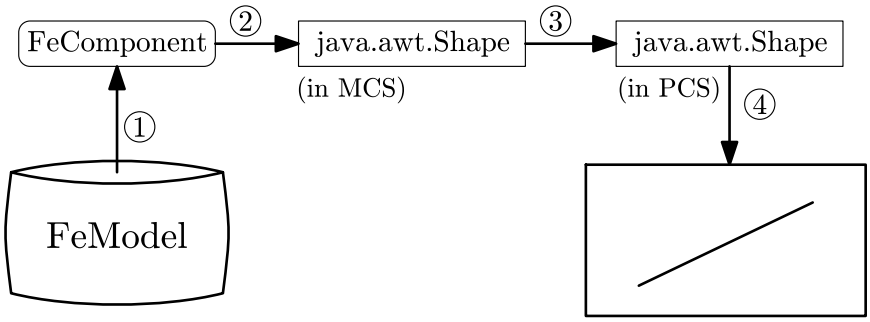


Fig. 4.14: Paint procedure

4.5 Interacting with a Visual FEM Model

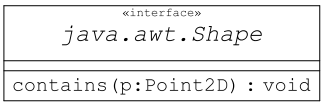
The previous section explained how an FeModel can be visualized. The next step is to add functionality to select FeComponents by picking them on the panel. Providing this functionality allows the user to obtain more information about a particular component. This section explains how picking can be supported.

4.5.1 Support from Java interfaces

The following two interfaces support the implementation of picking functionality:

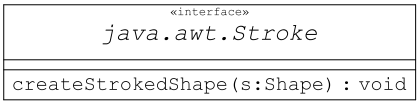
Shape

The `Shape` interface allows the evaluation of the location of a `Point2D` object relative to the boundary of a `Shape` instance. The `contains(Point2D p2d)` method returns *true* if the Point2D is located inside the `Shape`, otherwise it returns *false*.



Stroke

The `createStrokedShape( Shape s)` method of the `Stroke` interface returns an outline `Shape` which encloses the area that should be painted when the `Shape` is stroked according to the rules defined by the object implementing the `Stroke` interface.



4.5.2 Updating FemPanel to support picking

The following minor changes to the `FemPanel` allows a user to select FeComponents using a pointing device, e.g. a mouse.

Pick stroke

An attribure `m_pickStroke` of type `Stroke` is introduced. This attribute is used to create the snap zones of the individual components displayed on the `FemPanel`. If a `BasicStroke` instance is created with a pre-defined thickness of say 10 pixels, the snapzone resulting from this pick stroke would be 10 pixels wide. This implies that if a point is closer than 5 pixels on either side of a displayed shape, the shape's snap zone would contain it. The snap zone is defined in the PCS, not in the MCS. This ensures that the snap distance to a specific component is independent of the zoom factor and remains constant, e.g. 5 pixels. Thus, in model areas with a high component density, the picking accuracy is increased by zooming in.

Mapping from snap zone to FeComponent

A map called `m_shape2Obj` is added as an attribute of the `FemPanel`. Each key-value pair contains a `Shape` instance as key and an `FeComponent` as value. This map is maintained by the `paintComponent(Graphics g)` method.

Modification of `paintComponent(...)` method

Each time this method is invoked, the map `m_shape2Obj` is cleared. The actual drawing procedure remains the same as before, with the only extension

that the shape representing the FeComponent on the panel (`_shapePCS`) is stroked by the `m_pickStroke` to create a snap zone for this shape. This is done by invoking the `createStrokedShape(Shape s)` with _shapePCS as parameter. The object returned from this method is called a snap zone shape. This shape (_snapZone) is mapped to the corresponding FeComponent.

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
  m_shape2Obj.clear();
  Iterator iter = m_model.iterator(new TypeFilter(TrussElement.class));
  while(iter.hasNext()){
    TrussElement te = (TrussElement)iter.next();
    double x1 = te.node(0).getX();
    double x2 = te.node(1).getX();
    double y1 = te.node(0).getY();
    double y2 = te.node(1).getY();
    Shape _shapeMCS = new Line2D.Double(x1,y1,x2,y2);
    Shape _shapePCS = m_transform.createTransformedShape(_shapeMCS);
    Shape _snapZone = m_pickStroke.createStrokedShape(_shapePCS);
    m_shape2Obj.put(_snapZone,te);
    g2d.setColor(new Color(0,0,255,50));
    g2d.fill(_snapZone); // fills the snapzone
    g2d.setColor(Color.BLACK);
    g2d.draw(_shapePCS);
  }
}
```

Listing 4.3: paintComponent(Graphics g)

MouseListener

The actual picking is done by a `MouseListener` implementation. A simple example of such a listener is shown in listing 4.4. This `MouseListener` was implemented as an inner class inside the `PickableFemPanel` to keep the complexity of the example low. Whenever this `MouseListener` receives a mouse event, it iterates over the keyset of the `m_shape2Obj`-map and invokes the `contains(Point2D p)` method of each snap zone shape. The parameter passed to this method is the `Point2D` instance that represent the location where the `MouseEvent` occured. If a snap zone contains this point, the related FeComponent is retrieved from `m_shape2Obj`. This example only prints the picked FeComponent on the output console.

```
class Picker extends MouseAdapter{
  public void mousePressed(MouseEvent e) {
    Iterator iter = m_shapeToFeObjMap.keySet().iterator();
    while(iter.hasNext()){
      Shape s = (Shape)iter.next();
      if(s.contains(e.getPoint())){
        System.out.println(m_shapeToFeObjMap.get(s)
            + " was selected!");
      }
    }
  }
}
```

Listing 4.4: MouseListener implementation

4.5.3 Example

An example of a truss is shown in Figure 4.15. This example conveys the methods that were explained in this section. The `TrussModel` that is used is shown in listing 4.5. The model is created by extending the `aho.fem.FeModel` class and instantiating model component instances inside the constructor. As a result the FeObjects are created and added to the model, which is convenient for testing purposes.

The truss elements are represented as `Line2D` instances and the snap zones are indicated as shaded areas in figure 4.15.

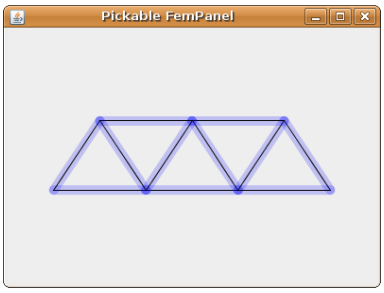


Fig. 4.15: Pickable FemPanel

If a mouse-click occurs inside a shaded area, the corresponding FeComponent is printed in the console. There are certain places where snap zones overlap. This implementation returns all the FeComponents if a mouse-click occurs in more than one snap zone.

```
package thesis;

import aho.fem.FeModel;
import aho.fem.Kernel;
import aho.fem.component.Node;
import aho.fem.component.TrussElement;

public class TrussModel extends FeModel{
  private static final long serialVersionUID = 1L;

  public TrussModel(){
    super();

    add(new Node("n1",new double[]{0,0,0}));
    add(new Node("n2",new double[]{2,0,0}));
    add(new Node("n3",new double[]{4,0,0}));
    add(new Node("n4",new double[]{6,0,0}));
    add(new Node("n5",new double[]{1,1.5,0}));
    add(new Node("n6",new double[]{3,1.5,0}));
    add(new Node("n7",new double[]{5,1.5,0}));

    add(new TrussElement("t01",new String[]{"n1","n2"}));
    add(new TrussElement("t02",new String[]{"n2","n3"}));
    add(new TrussElement("t03",new String[]{"n3","n4"}));
    add(new TrussElement("t04",new String[]{"n5","n6"}));
    add(new TrussElement("t05",new String[]{"n6","n7"}));
    add(new TrussElement("t06",new String[]{"n1","n5"}));
    add(new TrussElement("t07",new String[]{"n5","n2"}));
    add(new TrussElement("t08",new String[]{"n2","n6"}));
    add(new TrussElement("t09",new String[]{"n6","n3"}));
    add(new TrussElement("t10",new String[]{"n3","n7"}));
    add(new TrussElement("t11",new String[]{"n7","n4"}));

    Kernel.setModel(this);
```

```
35    }
  }
```

Listing 4.5: Fem Model of truss

4.6 Graphical manipulation of a FEM model

State-of-the-art finite element applications allow a user to create and manipulate finite element components using graphical input devices. Users prefer this way of interaction since it is more intuitive and the effect of actions is visible.

In the previous sections two aspects of FEM models and graphics were discussed, namely the *visualization* of FEM models and the graphical *manipulation* of FEM models. The first, i.e. visualization, is relatively simple and it belongs in the FEM domain of application development. Graphical manipulation of FEM models, however, requires an order of magnitude more development effort and actually belongs in the CAD domain of application development.

In essence, two approaches can be followed to achieve graphical manipulation of FEM models. The first approach would be to develop classes like the `PickableFemPanel` to support the creation and modification of finite element components via mouse input. For example, the coordinates of a mouse-click can be transformed to the MCS and used to create a node. This approach would quickly yield results, however all CAD functionality must be implemented again, which casts a shadow over the viability and judiciousness of the approach.

The preferred approach to achieve graphical manipulation of FEM models should be to use existing CAD applications and to develop an interface between the FEM model and the CAD system. Even if it takes more time initially to show results, eventual saving brought about by the re-use of existing CAD functionality will be significant. The viability of the approach is investigated from the perspective of CAD applications in the next section.

4.7 CAD manipulation of domain specific models

Historically, CAD systems developed independently from other engineering applications. CAD systems focused on how a user interacts with geometry while FEM systems, for example, focused on the numerical modelling of the physical behaviour of structural systems. Adding computer graphics to FEM systems allowed users to graphically specify FEM components and interact with FEM models. FEM software companies actually invested heavily into adding CAD-type front-ends to their systems to make the software more user friendly and accessible to structural engineers in general.

However, these front-ends are not used by other professions like architects. Consequently they neither support continuous information flow nor do they provide support for consistency of information.

Within the context of the research focus, the purpose of this section is to investigate how an existing CAD system can be adapted to serve as a graphical front-end for an existing FEM framework. This concept is generalized so that an existing CAD system can be used to manipulate geometric aspects of grahpical user interfaces for any model that involves geometry, e.g. facility management models, load-take-down models, etc.

The extention of CAD system functionality to manipulate domain specific, geometry-dependent models will contribute significantly towards developing usable software for industry. Within the context of this dissertation, however, it serves as a basis to establish bindings between diverse models.

4.8 CAD system architecture

A prudent architecture for CAD systems comprises the three layers shown in figure 4.16.

Input layer This layer is closely connected to the user interface. It creates and forwards commands to the command layer. Different GUI components, e.g. buttons, menu items or text input, can create the same underlying command. In this way the GUI is clearly separated from the rest of the CAD system, thus it can be customized without affecting other parts of the system.

Command layer This layer encapsulates the database layer. It interprets the commands created by the input layer and then executes the commands on the database. It also manages undo and redo functionality.

Database layer This layer contains the information of the CAD model instance, e.g. a set of CAD components, the selection statusses, meta data concerning the model, etc.

4.8.1 Interaction requirements

Geometric information in CAD models is encapsulated in CAD components. Consequently any interaction between CAD systems and geometry-dependent domain models requires the manipulation of CAD components. The functionality to do this has to be provided by the CAD system. Specifically, the following two requirements must be met:

Clearly defined interfaces

The CAD system should have well defined interfaces that specify the functionality of individual CAD components. This allows the addition of new

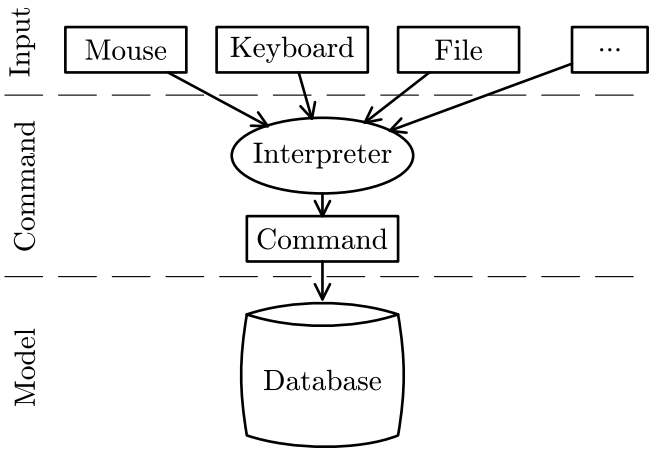


Fig. 4.16: Layers of a CAD system

components to the system, as well as interaction with existing components.

Extensible command language

An extensible command language and interpreter should be available. This allows a user to add functionality to manipulate the CAD components and the database.

4.9 Linking domain models to CAD applications

The primary purpose of linking a geometry-dependent domain model to a CAD application is to gain access to the geometry manipulation functionality of the CAD application. The extent to which the CAD application provides a front-end to the domain model is a matter of choice. The issue is complicated somewhat by the fact that certain operations on the domain model may require a combination of CAD functionality and pure domain functionality. Consider, for example, the assignment of cross-sectional properties to elements of a truss. Cross-sectional properties reside in the domain model and do not have a direct bearing on geometry. However, CAD applications provide support for picking elements graphically, which is very useful to correctly assign properties to specific elements of the truss. In such a case it may be useful to extend the link between the domain model and the CAD application to support such assignments. However, the crucial aspect when linking a geometry-dependent model with a CAD application is that the different models should remain independent from one another.

A successful link has the following properties:

- Existing functionality and operation of the CAD application remain

the same. Additional components and functionality are added as required to provide the linking with the domain model. Any CAD model instance can exist independently of the added components and functionality and without the domain model being present.

- Existing functionality and operation of the domain application remain the same. Additional components and functionality are added as required to provide the link with the CAD model. Any domain model instance can exist independent of the added components and functionality and without the CAD application being present.

The required functionality, as well as the clear separation of the models can be achieved through the implementation of the proxy-based design pattern described below.

4.9.1 Proxy-based interaction design pattern

New CAD components that are developed to provide the link with the domain model should be proxy objects, denoted as CAD proxy components. These objects implement the interfaces of the CAD application that are necessary to provide the CAD functionality required. The CAD proxy components do not store the geometric data which is important to the domain application directly within itself. Instead each proxy instance holds a reference to the domain application object which it represents, and the geometry data is stored in the domain object. As a result all the functionality of the CAD application becomes available to manipulate data which in fact resides in the domain application. Some new CAD commands may also be implemented, e.g. to allow the creation of the CAD proxy components. The design pattern is shown in figure 4.17.

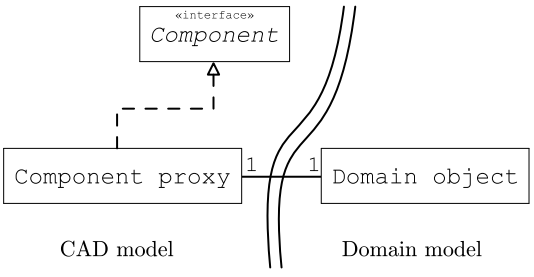


Fig. 4.17: Design pattern

The crucial separation between the two models is achieved in a way which is transparent to the user. The CAD system is not aware that it actually operates on a domain model and each of the CAD and the domain application can exist without the other. Figure 4.18 displays the intersection between the CAD system and the domain application graphically. The grey

part, i.e. the intersection, contains the CAD proxy objects and the related newly created CAD commands. This linking-part is transient, i.e. it is created when a domain model is loaded into the CAD system and discarded when the working session ends. When linking is not required, the CAD and the domain model function independently.

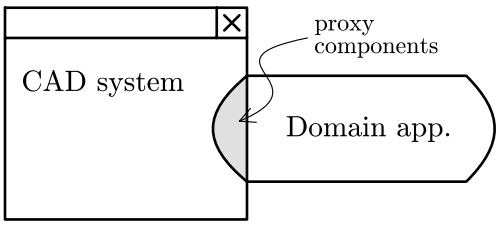


Fig. 4.18: Intersection: CAD and domain applications

4.9.2 Implementation example

The CAD system CADEMIA [Firmenich 2006; CADEMIA 2007] is used to prove the design pattern described above. CADEMIA is an open source software system developed at the Bauhaus-Universität Weimar, Germany. It meets the requirements set out in section 4.8 and its openness makes it an attractive CAD system to use in this test example.

Consider a truss that comprises of `FemNode`[2] and `FemTruss`[3] components only, similar to the example in section 4.5.3. The aim now is to provide full CAD functionality in order to support the graphical manipulation of the FEM components via an existing CAD application. The following two CAD proxy components are needed as well as CAD commands to create them:

CadNode

A `CadNode` wraps a `FemNode` instance. The only attribute of the `CadNode` instance is the `FemNode` that it wraps. The coordinates are stored inside the wrapped `FemNode`. If the `transformBy(AffineTransform af)` method is called, the `CadNode`-implementation of this method modifies the coordinates of the underlying `FemNode` object. The graphical representation of the `FemNode` is taken care of by the `CadNode` instance, i.e. the representation is created based on the location of the wrapped `FemNode` whenever the CAD system asks for it.

[2] A `FemNode` contains the degrees-of-freedom which are solved to determine the structural behaviour of the truss.

[3] A `FemTruss` instance is a finite element component that has the functionality to calculate its own element stiffness matrix, $[K_e]$, which is then used to assemble the system stiffness matrix $[K_s]$ of the truss in order to model the structural behaviour of the truss as a whole.

The new command, `AddNode`, creates firstly a `FemNode` instance and then invokes the constructor of the `CadNode` class, passing the created `FemNode` as argument to it. The command adds the `CadNode` to the component set of the CAD system and the `FemNode` instance to the `FeModel` instance.

CadTruss

In the same way that the `CadNode` wraps a `FemNode` instance, a `CadTruss` wraps a `FemTruss` instance.

Figure 4.19 shows the object references (⟶) between the three objects in the FEM model (two nodes and one truss element), as well as the object references between the CAD proxy objects and the underlying FEM objects. The geometry of the `CadTruss` component depends on the location of the two `FemNode`s at the endpoints of the `FemTruss` object. When the CAD system requests the shape of the `CadTruss`, the `CadTruss` component creates a shape based on the coordinates of the underlying `FemNode`s. It obtains these coordinates via the `FemTruss`. The `FemTruss` only contains topological information, i.e. the references of the `FemNode`s that define its geometry, and obtains the geometrical information from the `FemNode` objects.

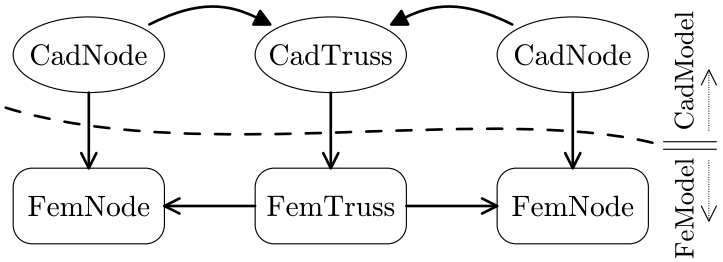


Fig. 4.19: Object relationships

In the CAD application the topological dependency between the `CadTruss` and the `CadNode`s is modelled using a binding mechanism. These bindings are represented as curved arrows in figure 4.19. This is done to ensure that graphical representation of the `CadTruss` updates correctly if the `CadNode` is modified[4].

The `AddTrussElement` command does the following:

1. It receives two `CadNode` instances.
2. It creates a `FemTruss` between the two `FemNode`s wrapped by the `CadNode`s.
3. It creates a `CadTruss` by wrapping the `FemTruss` objects created in the previous step.

[4]Note that if the complete view is redrawn after each user action, the binding mechanism to keep the CAD view consistent with the underlying FEM instance is not required. However, recalculating the view is an expensive operation which should be minimised in order to maintain the responsiveness of the application

4. It establishes the bindings between the `CadNode`s and the `CadTruss` to keep the view consistent.
5. It adds the objects created in steps 2 and 3 to the respective data structures i.e. the CAD component set and the FEM model.

Consistency issues: Assume that the coordinates of the `CadNode` are modified. This modification is performed directly in the underlying `FemNode` because the `CadNode` instance wraps the `FemNode`. This information cannot become inconsistent because it is only stored in one place, namely inside the `FemNode`. The same applies to the `FemTruss`, it always obtains the coordinates of its end points directly from its connected `FemNode`s.

However, the connections between the `CadTruss` and the two `CadNode`s are not modelled directly because no connections exist between these components in the CAD environment. The relationship between these components is modelled with the binding mechanism. After the modification of the `CadNode`, the FEM model itself is consistent, but the CAD view does not reflect the actual FEM model state because the `CadTruss` is unaware of the change to the `CadNode` component and therefore not updated. The binding mechanism detects that the `CadNode` was modified, and then updates the graphical representation of the `CadTruss` to restore the consistent state between the view of the FEM model and the FEM model itself.

Summary: The issue of supporting consistency in the flow of geometric information from a CAD application to a FEM application was discussed. For the case of structural engineering it was argued that an additional model, namely the Structural Engineering Model (SEM) is required to merge structural engineering concepts with CAD-geometry before any attempt at supporting consistency can be made. Technical aspects of FEM and CAD applications were then analysed and it was shown that, provided some simple requirements are fulfilled, a CAD application can be extended to supply geometric information to an engineering domain application like FEM. More importantly, it can also manipulate the geometry and effectively become a graphical pre-processor for the domain application. The requirements and the software design pattern to achieve the said functionality, without losing standalone capability in either application, was discussed and demonstrated. These concepts are developed into a solution approach for the integration of specialized domain models in the next chapter.

Chapter 5

Concepts for the integration of specialized engineering models

From the engineering point of view, specifically from the point of view of the structural engineer, the common ground between CAD applications and other engineering domain applications like FEM is the geometry and annotations contained in the CAD models.

In this chapter a middleware model like the SEM is proposed as a general solution for transferring CAD information to engineering applications. Furthermore, the focus is on supporting the *consistency* of the information flow, for which a solution based on inter-model bindings is proposed. The specific case of CAD-FEM applications, with the SEM as middleware model, is used to clarify the basic assumptions and to provide examples of certain concepts. Hence the role of the Structural Engineering Model (SEM), introduced in section 4.2, is extended. It is important to note that although the specific case of structural engineering is dealt with here, the solution approach is valid for any geometry-dependent engineering application.

5.1 Basic approach

The proposed concept for the integration of engineering models is based on two simple, but crucial, fundamental assumptions, which may be summarized as follows:

- Engineering tasks are executed by various people. Each participant has a different role and responsibility and typically uses a specific software model in the execution of his task. This work pattern must be adhered to. This implies that engineering software development should not aim at enabling, for example, a draftsman to perform structural engineering

tasks. The aim should only be to support the various participants in working together effectively.

- The engineering models used by the various participants have to remain operative as seperate, standalone models. This implies that it is assumed that engineering applications will continue to be developed independently, at least in the medium term. The aim is to use the different models as coherently as possible.

Below, the specific case of structural engineering is analysed within the context of the assumptions. Such an analysis is required for any engineering application for which a solution is to be created.

5.1.1 Draftsman and Structural Engineer

A draftsman is a person who works for the structural engineer and whose task is to create drawings that are used in the design and construction of a building. The structural engineer is responsible for the design of the building. These two disciplines work in parallel during the execution of the building design. A single person can perform both tasks if the scale of the project permits it.

Each task has information sources (input). It then processes the information (execution), and provides some results (output). These three aspects are briefly discussed to highlight the similarities as well as the differences between the tasks.

Drafting task

Different drafting tasks occur throughout the design phase. The output of a drafting task is a drawing or model. When models are created the final product is again drawings cut from the model. The following drawing types are typical for structural design:

- Concrete layout drawings
- Concrete reinforcement drawings
- Structural steel layout drawings
- Structural steel connection drawings

The input depends on the type of drawing that is produced by the drafting task. If the resulting drawing is a concrete layout drawing, the input is typically drawings that were created by an architect. In contrast, a concrete reinforcement drawing is based on the reinforcement design of the structural engineer, as shown in figure 5.1.

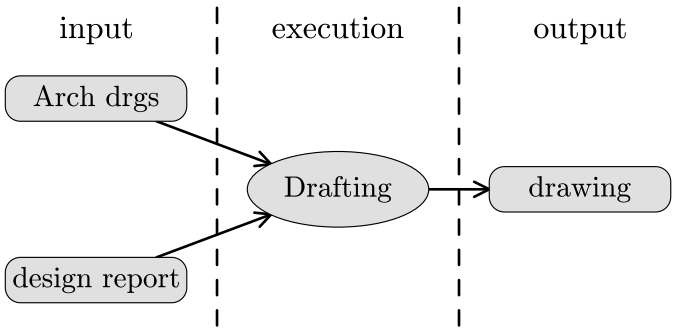


Fig. 5.1: Drafting task

Computer tools for draftsmen

It is important to note that the output of the drafting task is always a drawing of some kind although the input sources may vary substantially. Consequently draftsmen execute their task with tools that enable them to create high quality drawings efficiently. Whether this tool is a traditional 2D CAD system or a modern 3D parametric modeler, the end result remains drawings, usually printed on paper, that communicate specific aspects of the building.

Design task

Different design tasks are executed throughout the design phase of the building. The following tasks are typical in standard building design:

- Global stability analysis and design.
- Structural steelwork design.
- Foundation design
- Detail member design
- Connection design

The documentation of a specific design task is crucial in communicating the design to other team members and to the authorities.

The design is documented in a design report which also contains design drawings or sketches. Thís report is used as input for the drafting task. Some authorities require the approval of the design and without a well structured design report it becomes an extremely difficult task to verify the design of the building.

Unlike the drafting task, where only the result is of importance, the decision making and design intent must also be captured in the design phase.

Computer tools for structural design

Computer software used by structural engineers during the design task vary from spreadsheets, word processors and CAD systems to FEM and design applications. A traditional CAD system, however, does not support the work of the structural engineer the same way that it supports the work of the draftsman because it lacks the ability to capture structural design concepts, intent and numerical analysis parameters.

Interaction between design and drafting

The interaction between the design and drafting tasks is dynamic. Sometimes the design task requires information from the drafting task and vice versa, as shown in figure 5.2.

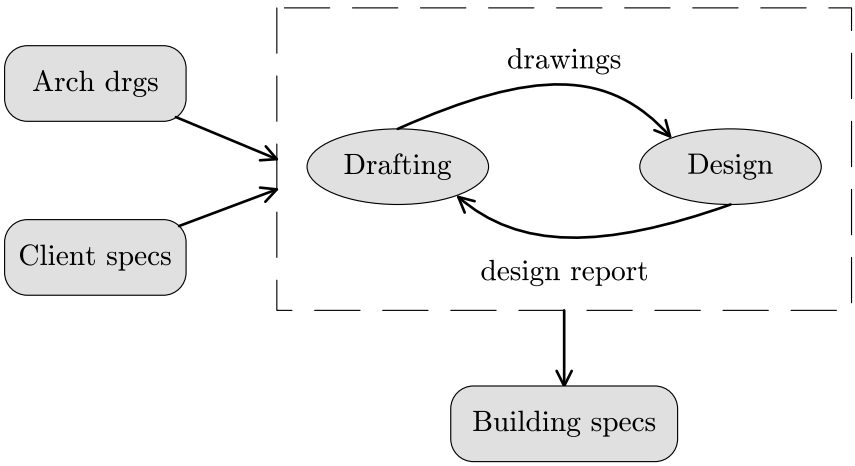


Fig. 5.2: Iterative nature of design and drafting tasks

Information is thus frequently exchanged between these two tasks. However, the nature of the two types of information is different. The drawings coming from the drafting task contain factual data regarding geometry as well as annotations, while a large part of the report coming from the design task is in the form of statements and instructions that have to be interpreted by the draftsman. Also, not all the information in the design task is required by the drafting task. For example, the drafting task does not depend on the structural analysis itself but only on the interpretation of the results thereof by the engineer. The point of information exchange where the vast majority of problems arise is where geometric information flows from the drafting task to the design task, and it is for this point that a solution is proposed.

In coherence with the fundamental assumptions of the solution approach, the aim is to utilize applicable information in the CAD models, namely geometric information, in the development of the FEM and design models, in a

way that allows all participants to work effectively. Furthermore, supporting effective information flow from the CAD model to the FEM model should not prevent either application from operating fully and independently on its own.

5.2 CAD-Eng middleware model

The software technology needed to achieve CAD interaction with engineering domain applications was discussed and developed in section 4.7. A design pattern was proposed in which the engineering components *wrap* the geometric information that is manipulated by the CAD application. An example implementation was developed which combined the CADEMIA system with a truss analysis application. It was shown that the necessary interaction, as well as the essential separation of the two models is achieved by the technology as proposed.

In structural analysis, truss models, however, are discrete models (see section 4.1.1) which represent a special case in which it is possible to map CAD objects almost directly to truss objects. In the general case, namely that of continuous models (see section 4.1.2), a direct mapping is not possible and an additional model is required that encapsulates both the geometric information of the CAD side and the semantics of the structural engineering side. For this case the Structural Engineering Model (SEM) was introduced in section 4.2.

Similar to the case of continuous structural engineering models, the components of geometry-dependent engineering applications are in general not directly mappable from CAD components. An additional model, denoted as the CAD-Eng middleware model, is required. The components of the CAD-Eng model interact with both the CAD and the engineering application's components, hence the term middleware. The CAD-Eng components provide for the geometry, as well as the semantics of the engineering application and it has methods to spawn an instance of the engineering model. Hence the CAD-Eng model may be viewed as a pre-processor of the engineering model whose geometric components are derived from, and manipulated by, the CAD application.

The interaction between the CAD application, the CAD-Eng middleware model, and the engineering domain application is explained with the aid of figure 5.3:

A model instance of each participant is shown. The CAD model instance contains two types of components, namely the native CAD components, as well as proxy CAD components. Some of the native CAD components are used to create middleware components, specifically those middleware components that are geometry-dependent. In the process the necessary geometric data are transferred by value from components of the CAD model to com-

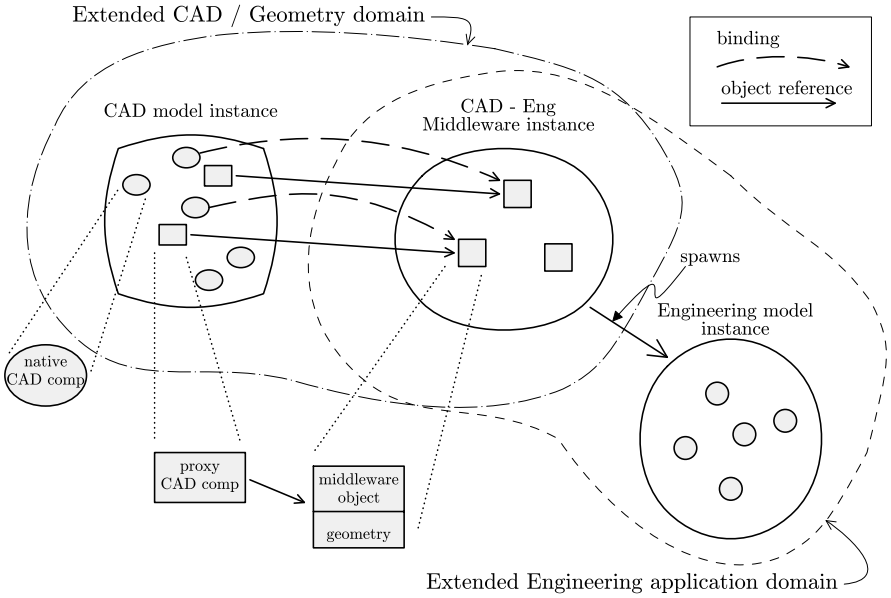


Fig. 5.3: CAD-Eng middleware

ponents of the CAD-Eng model. A proxy CAD component is instantiated whenever a CAD-Eng middleware component is created. Each proxy CAD component holds a reference to a CAD-Eng middleware object. The middleware object contains the geometric information which the proxy CAD component manipulates. In this way the proxy components make CAD functionality available to objects that actually reside in the CAD-Eng middleware model. Together with the CAD model, the middleware model forms an extended CAD/Geometry domain. In addition to geometric information, the middleware model supports the semantics of the specific engineering model. Once the CAD information has been transferred to the middleware model, and, if necessary, manipulated using the proxy CAD components, the CAD-Eng model spawns an engineering model instance which is used by the engineer in executing his tasks. Consequently the CAD-Eng middleware model, together with the engineering domain model, forms an extended engineering application domain.

For the case of FEM applications of structural engineering, the CAD-Eng middleware model is a Structural Engineering Model (SEM). A SEM contains structural components like beams and slabs, supports, lineloads and arealoads, etc. Consider the case where the structural engineer wants to use a line in a CAD model to represent a beam in his FEM model. He would then select the line and execute a command to create a SEMbeam using the selected native CAD line. During the creation of the SEMbeam, the geometry of the line is transferred by value to the SEMbeam, and a

CADbeam proxy object is created in the CAD application. The CADbeam references the SEMbeam and provides the required functionality of the CAD application to manipulate the geometry contained in the SEMbeam. The CADbeam is typically displayed in the CAD application using a special shape that differs from the shape used to display the line. However, these two objects can be manipulated individually. The line is owned by the draftsman, the CADbeam by the structural engineer, and each should be manipulated only by its owner.

Once a CAD-Eng model has been developed for a given CAD application and a given engineering application, effective transfer of geometric information from the CAD to the engineering application, as well as graphical manipulation of the engineering objects, can be achieved without violating the fundamental assumptions of the solution approach. It should be noted that a CAD-Eng model can be developed for any engineering application that is geometry dependent. Apart from providing seamless transfer of geometric information from the CAD to the engineering domain, it is postulated that the developers and the users of engineering applications can benefit by utilizing the inherent grahical manipulation capabilities of CAD systems as geometry pre-processors for their applications.

5.3 Consistency problem

The CAD-Eng model described above offers a solution for the transfer and manipulation of geometric information. However, the problem of consistency of the information is not addressed. As a result of the fundamental assumption that the CAD and engineering applications must remain clearly separated, standalone applications, geometric information is stored in both the native CAD components and the CAD-Eng components and both can be manipulated individually, typically by different participants of the engineering project. Consequently the two sets of information can become inconsistent. It is important to note that exactly this problem of inconsistency between CAD information and information used in engineering domain applications is well known and occurs daily in current engineering practice, it is not caused by the solution approach under discussion.

However, the introduction of CAD-Eng middleware provides the potential to propagate geometric changes, since each CAD-Eng component depends on one or more native CAD components. This means that all geometric changes that matter can be forwarded from the changed native CAD components to their dependent CAD-Eng components. It is then up to the CAD-Eng model and its owner to deal with the changes in a way that makes the CAD-Eng model consistent with the CAD state again. In this way consistency downstream of the CAD model can be achieved. Upstream consistency, i.e. from the engineering application back to the CAD model,

is not addressed in this dissertation.

The propagation of CAD-model changes to the CAD-Eng model is achieved with the aid of inter-model bindings, described below.

5.3.1 Inter-model bindings

All geometry-dependent components in the CAD-Eng model are based on native CAD entities. For example, if a floor slab is required in a SEM, a number of CAD lines and polylines that describe the slab's boundaries have to be selected, after which they serve as input in the creation of the floorslab. This implies that all changes to the source CAD entities can be forwarded to the dependent CAD-Eng entities, i.e. a binding relation between the two sets of entities can be established. The mathematics of the binding relation is discussed in chapter 6.

The bindings between components of the CAD model and the CAD-Eng model are shown graphically in figure 5.4. During construction of any geometry-dependent CAD-Eng component, a binder-object is created that references both the source CAD component/components and the dependent CAD-Eng component. The binder-objects reside in the CAD-Eng model.

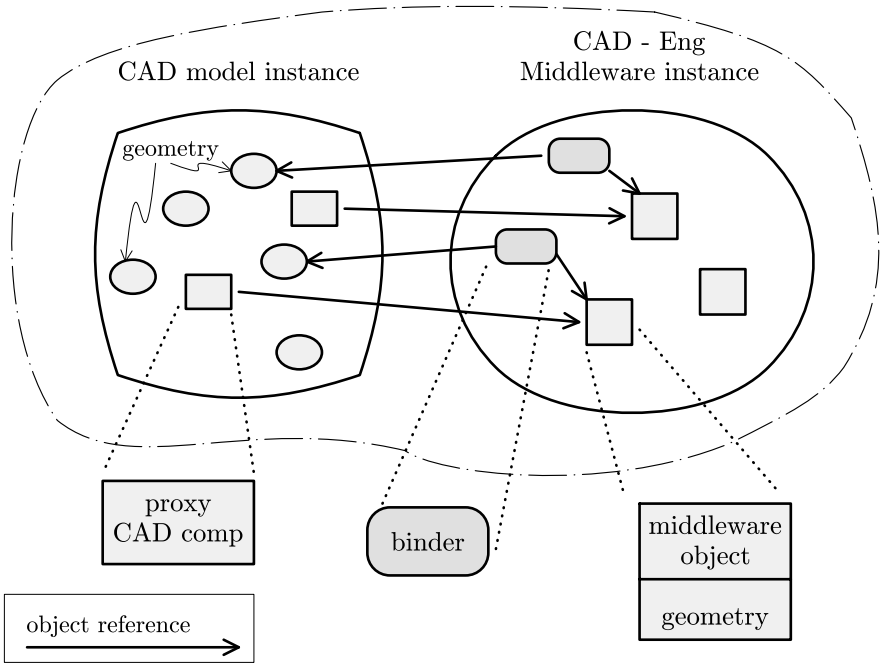
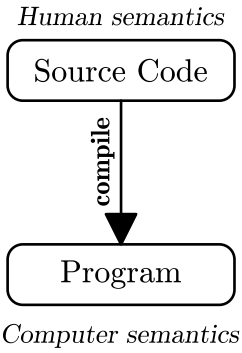


Fig. 5.4: Inter-model bindings

5.3.2 Dealing with changes

At a suitable, user-controlled time, the set of binder-objects in the CAD-Eng model can be traversed to find all instances of changed CAD components. Any technique for recognizing changes, e.g. versioning, can be used to identify changed components. The changes are then forwarded to the dependent CAD-Eng components, where they have to be dealt with. The effect of changes is highly dependent on the type of engineering component and typically requires input from the engineer.

However, much of the routine work associated with change management can be automated. For example, once the engineer has noted and accepted the changes of lengths of lines outlining a floor slab, the slab can be re-meshed automatically. Once the changes have been dealt with in the CAD-Eng model, a new instance of the engineering application model is spawned by the CAD-Eng application.



Dealing with changes in the CAD-Eng model and then spawning a model for the engineering application is similar to the way changes are effected in the software industry. A programmer produces source code that is not usable by a computer until it has been compiled. The easiest way to modify the compiled code when changes are required, is to change the source code and to recompile the program. In the same way that the programmer spends time on modifying the source code and not the compiled code, the engineer spends time on modifying the CAD-Eng model, and not the derived engineering model.

In chapter 2, figure 2.2 presented a graphical view of the research focus, namely examining the interface between CAD and FEM models to provide better integration between these two domains. This interface now changes with the introduction of the specific CAD-Eng middleware model, i.e. the structural engineering model as shown in figure 5.5.

5.4 Summary

The solution approach for consistent transfer of information from CAD applications to engineering applications was described in general terms in this chapter. Specific references were made to the Structural Engineering Model (SEM), which is the CAD-Eng middleware model proposed for combining CAD and FEM applications. More details of the SEM and how it is used is presented in chapter 7, “Working with Structural Engineering Models”.

The solution approach as described requires the application of mathematical constructs and algorithms of the algebra of relations and graph theory. The necessary mathematical background is presented in chapter 6.

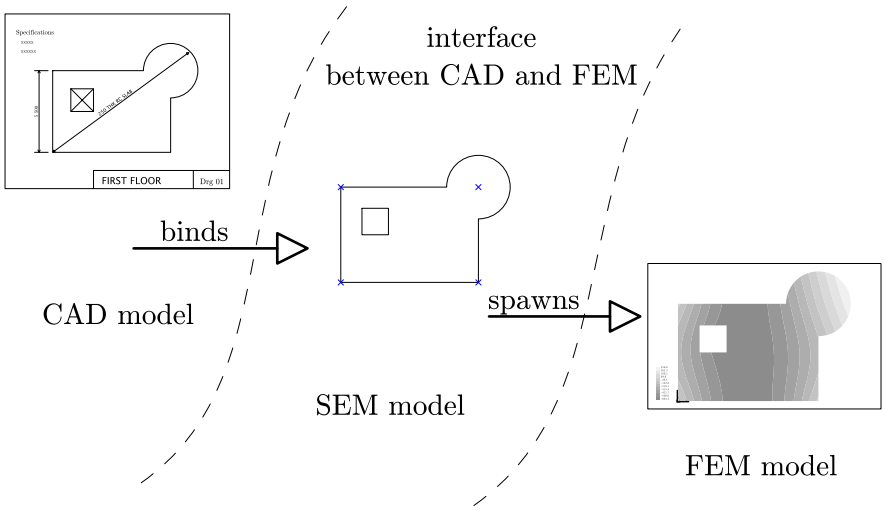


Fig. 5.5: Specific middleware model to bridge the gap between CAD and FEM domains

Chapter 6

Bindings and updating

The consistency of information that is transferred between engineering domain applications can be supported with the aid of inter-model bindings as described in chapter 5. The techniques and technologies required to implement the consistency support is discussed in this chapter.

Inside a model bindings may be used to represent certain dependencies and these are called intra-model bindings. By extending this concept, dependencies between two different models are represented by inter-model bindings. The fundamentals of bindings are described and the intra-model and inter-model binding relations are formulated mathematically. The binding relation formed by the unification of the intra-model and inter-model binding relations is introduced and denoted as the unified binding relation. The unified binding relation is required during updating. Whenever reference is made to the binding relation, the unified binding relation is implied.

Updating is the process whereby the binding relation is used to actualize the state of the dependent model so that it becomes consistent with the state of the source model. It is important that the owner of the source model ensures that his model is internally consistent before handing it over to the owner of the dependent model for updating. The updating mechanism and a software design pattern for implementing it is described. The binding relation not only determines which objects require updating, it also controls the sequence in which the updating has to be executed. A novel algorithm for detecting the updating sequence is described as well as its mapping to an object-oriented computer model. The performance of the algorithm and its implementation is compared to that of an existing algorithm. This is important since graph calculations are often computationally expensive.

6.1 Definition of binding relation

A binding is defined by [Pahl and Beucke 2000] if one of the following situations occur. 1) a method of object A changes an attribute of object B or

2) a method of object B uses an attribute of object A. Figure 6.1 represents a single binding. Obj_A is the binding object whereas Obj_B is the bound object. Obj_A binds Obj_B and Obj_B is bound by Obj_A. The arrow (→) between these two objects represents the binding. If Obj_A changes then Obj_B should be updated to maintain consistency, thus Obj_B depends on Obj_A.

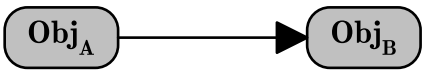


Fig. 6.1: Binding relationship

A binding relation is a set of object dependencies. The relation may be contained inside a single model, or it may span the boundaries of a model as presented in the previous chapter.

6.1.1 Intra-model binding

Let M be the set that contains all the objects of a model. The binding relation B of this model is then defined as the set that contains all the ordered pairs of the Cartesian product of $M \times M$ where the end vertex of the ordered pair depends on the start vertex of the ordered pair.

$$B_{\text{intra}} := \{(a, b) \in M \times M \mid b \text{ depends on } a\} \tag{6.1}$$

A useful application of intra-model binding inside a CAD model is the introduction of dimensions that are bound to the geometry that it describes.

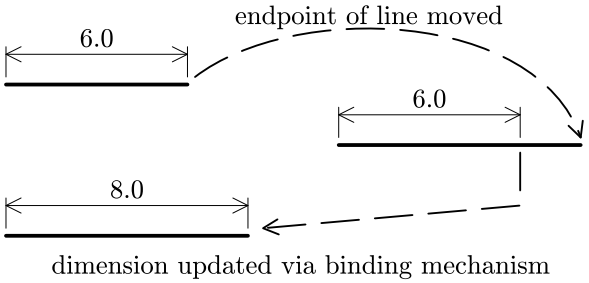


Fig. 6.2: Intra-model binding

Figure 6.2 shows the behaviour of a dimension component that is bound to the line that it dimensions. The endpoint of the line is moved two units to the right. Without a binding, the inconsistency that is created between the line and the dimension component must be corrected manually. Assuming that the binding mechanism detects the change in the line component, it can initiate the update of the dimension component. This specific problem is

traditionally solved inside various CAD systems in a specific way - generally called associative dimensioning. Here a general approach is developed that can handle associative dimensioning as one specific requirement but that is totally general in its formulation. Inside a Structural Engineering Model, for example, a support may be bound to a point on a beam. When the geometry of the beam changes, the position of the support can be updated with the aid of the binding mechanism. Change detection and updating are discussed later in this chapter.

6.1.2 Inter-model binding

The single model environment can be extended with the definition of set M as the set that contains all the objects of the source model and set N as the set that contains all the objects of the dependent model. The binding relation between these two models B is now defined as the set that contains all the ordered pairs of the Cartesian product of $M \times N$ where the end vertex of the ordered pair depends on the start vertex of the ordered pair.

$$B_{\text{inter}} := \{(a, b) \in M \times N \mid b \text{ depends on } a\} \tag{6.2}$$

6.1.3 Unified binding relation

When a dependent model has to be updated to become compatible with a source model, the inter-model bindings between the source and the dependent model, as well as the intra-model bindings in the dependent model have to be accounted for. For this purpose the unified binding relation is defined:

$$B_{\text{unified}} := B_{\text{intra}} \cup B_{\text{inter}} \tag{6.3}$$

in which B_{intra} is the intra-model binding relation and B_{inter} is the inter-model binding relation.

6.1.4 Multi-model binding and updating

The unified binding relation can be extended to account for any number of source models and any number of dependent models. In such a case the unified binding relation is the unification of all the inter- and intra-model binding relations. An example [Firmenich 2002] with 3 models and 3 inter-model binding relationships, indicated by the arrows, is presented below.

Figure 6.3 shows three objects that represent a load-bearing column in three distinct models. The first object represents the geometry of the column inside a CAD model. The second object is part of another standalone model e.g. structural design model, in which it is used to calculate the amount of reinforcement required by the column to withstand the applied loading. The third object is responsible for the sizing and placement of the reinforcement

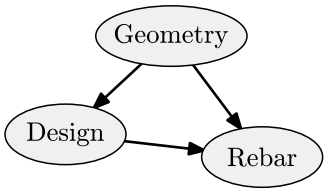


Fig. 6.3: Multi-model binding relation

so that it fits inside the column according to building regulations. This object is part of the reinforcement model.

If the geometry changes inside the CAD model, an inconsistent state arises between the geometry object and the other two objects that depend on the geometry. End-users must be assisted to update the dependent models to reflect the change in the geometry.

By simple engineering judgement it is clear that the two dependent objects should be updated in the order design object first, reinforcement object second. If the reinforcement object is updated before the design object, it will be consistent with the geometry object i.e. the layout of the reinforcement will fit inside the column *but* the amount of reinforcement in the column will be based on an outdated column design.

The update sequence can be computed if the unified binding relation is represented as a graph: the objects are the vertices and the dependencies between them are the edges of the graph. The update sequence can then be determined by performing a topological sort on the graph.(See section 6.5).

A successful updating mechanism depends on the following three fundamentals:

- The modelling of dependencies in a graph.
- The detection of changes and the update sequence.
- The way the system supports updating.

A discussion of each of these three aspects follows.

6.2 Graph modelling

A graph is a suitable mathematical structure for the modelling of bindings. The vertex set of the graph contains all the binding and bound objects and the edge set contains all the ordered binding-pairs. Different possibilities exist to map the graph to the computer [Turau 1996]. These possibilities are briefly discussed.

6.2.1 Boolean matrix

A boolean representation of a graph with n vertices consists of a $n \times n$ matrix that contains either a *true* or a *false* in each cell. If an edge exists between two vertices, the edge is represented as a *true* entry in the boolean matrix. The edge is located at the intersection of the row of the start vertex and the column of the end vertex. If the edge is undirected, two *true* entries are made in the boolean matrix.

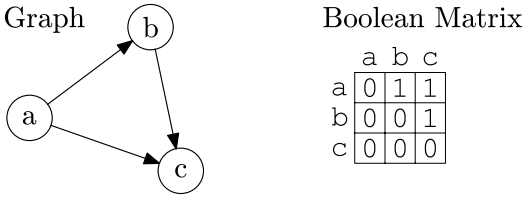


Fig. 6.4: Graph represented as a boolean matrix

A row contains all the outgoing edges of the vertex that it represents and a column contains all the incoming edges of the vertex corresponding to it.

Using the boolean matrix representation of a graph:

- Operations are performed by a sequence of matrix calculations.
- Scaling is a problem due to the allocation of storage for *all* possible edges and not only for edges that exist.

6.2.2 Adjacency lists

Adjacency lists are a more complicated data structure than a boolean matrix to map a graph to the computer, but it provides a solution that is more scalable.

Given Graph G

$$G := (V; E) \tag{6.4}$$

with vertex set V

$$V := \{a \mid a \text{ is a vertex } \} \tag{6.5}$$

and edge set E

$$E := \{(a, b) \in V \times V \mid \text{ edge from } a \text{ to } b\} \tag{6.6}$$

V is used as key set in a mapping that maps each vertex of V to an ordered pair (Γ, Λ).

The start vertex of the ordered pair is defined as

$$\Gamma := \{a \in V \mid \bigvee_{b \in V} (a, b) \in E\} \tag{6.7}$$

Likewise, the end vertex of the ordered pair is defined as

$$\Lambda := \{b \in V \mid \bigvee_{a \in V} (a, b) \in E\} \tag{6.8}$$

Thus, for a specific vertex in a graph, the start vertices of the incoming edges are stored in Γ and the end vertices of the outgoing edges are stored in Λ.

Figure 6.5 shows an example of how a graph can be mapped to the computer using adjacency lists.

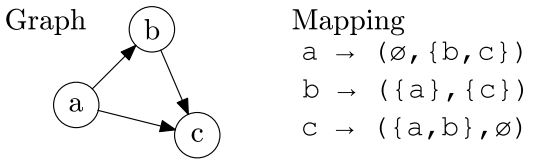


Fig. 6.5: Graph represented as adjacency lists

Although algorithms that operate on a graph represented by adjacency lists are more complex than before due to the repesentation of the graph, the scalability of this representation type is much better than the boolean matrix approach because only edges that exist are represented.

6.2.3 Edge Objects

The third way of modelling a graph in the computer is by representing each vertex in the graph as an object and mapping the graph edges to `Edge` objects. Each `Edge` instance references two objects, i.e. the mapped start and end vertices of the corresponding edge in the graph.

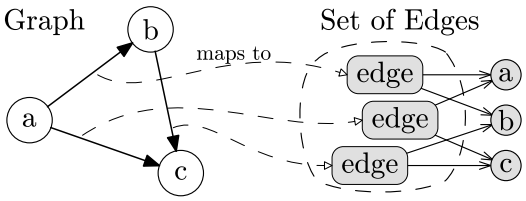


Fig. 6.6: Graph represented as a set of `Edge` objects

Figure 6.6 shows the same graph as before, now modelled by three `Edge` objects referencing their corresponding vertices. The solid line arrows (⟶) are used to represent object references.

This dissertation uses the third way of graph modelling due to its good scalability properties and secondly, that edges inside the graph are mapped to objects, thus graph-operations can be performed directly on sets that contain edges.

6.3 Change detection

Engineering work is characterised by a large number of relatively small changes that occur in between specific states of the work. This is sometimes refered to as 'deferred transactions'. It is neither practical nor advisable to update changes on a continuous basis. What is required is that updating takes place whenever the user decides that it is necessary. In accordance with this work pattern, the function of change detection is to determine which objects were changed since the last consistent state of the model(s). Two different approaches to change detection are possible. The first approach is that the binding mechanism gets notified whenever a change event occurs and assembles these changes until they are dealt with at update time. The second approach is that the binding mechanism looks for changed objects at update time.

6.3.1 Event driven change detection

A practical approach to detecting events is that the model provides a listener system where the binding mechanism can register as an event listener. This so called listener design pattern is frequently used in graphical user interfaces.

Gui example:

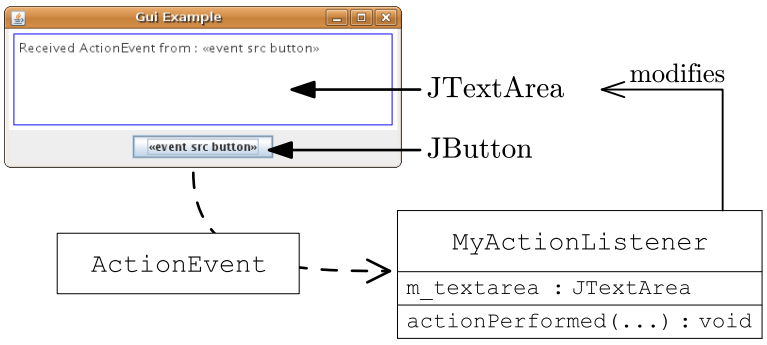


Fig. 6.7: Listener Design Pattern

Figure 6.7 shows a simple Java `JFrame` that contains one `JButton` and one `JTextArea`. If a user presses the button, a text line is added to the text

area. The underlying mechanism to support this functionality is part of the Java Swing components (the `JButton` and `JTextArea` are both Swing components). Each Swing component allows the registration of different listeners and listener types. In this example an `ActionListener` is implemented (i.e. `MyActionListener`) and registered at the button. This object has a reference to the `JTextArea` and appends a line each time its *actionPerformed(ActionEvent e)* method is invoked. Each time the button is pressed, all the `ActionListeners` registered at the button receive an `ActionEvent` via the invocation of the *actionPerformed(ActionEvent e)* method specified by the `ActionListener` interface. The `ActionEvent` encapsulates information about the event that took place, e.g. a reference to the component where the event occurred.

Observing the component set

Figure 6.8 presents the listener system for change detection. Two interfaces are required by this design pattern [Firmenich 2006]. The first interface, i.e. `ObservableSet`, is implemented by the container that contains the objects that are monitored. It provides methods to manage the list of `ObservableSetListener`s that monitor an `ObservableSet`.

The second interface, namely `ObservableSetListener`, allows an object to receive events fired by the `ObservableSet` if it is registered with the `ObservableSet`.

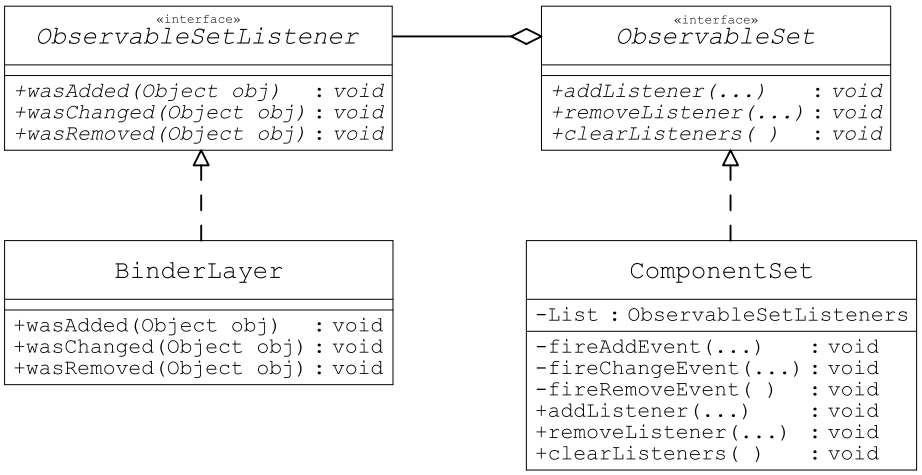


Fig. 6.8: Listener Design Pattern

The `ComponentSet` implements the `ObservableSet` interface. Although it is not specified by the interface, some convenience methods are part of the implementation of the `ComponentSet`. These methods are private methods and are called from somewhere else inside the class to support the firing of

an event if it occurs (e.g. *fireAddEvent()*).

The `BinderLayer` implements the `ObservableSetListener` interface. This allows the `BinderLayer` to receive events from the `ComponentSet` whenever objects are added, removed or changed.

The end result is that the `BinderLayer` gets notified each time an object is added to, removed from or changed inside the `ComponentSet`. When it is time to perform an update, the changed objects are known by the `BinderLayer` and the dependent objects can be found.

Advantages: It is not necessary to search for changed objects when it is update time.

The communication from the `ComponentSet` to the `BinderLayer` occurs over a well defined interface that is used by other systems inside the application as well, e.g. the updating of the user interface is also event driven.

Disadvantages: The `BinderLayer` must always be present in the parent application to catch the events and it must be available in the dependent application to support the update process. Sharing the `BinderLayer` between two applications in real-time requires either a dedicated network connection between the two applications or some client-server architecture to support parallel work.

It clearly places a burden on the source application, with no direct advantages for this application. Consequently there is no direct incentive for the developers of the source application to implement and maintain the additional software.

Application: Event-based change detection integrates well inside a single model environment where a binding mechanism is used to keep dependent information consistent i.e. intra-model consistency.

Intra-model binders are also denoted as *observer-based* binders in this dissertation, because they are utilizing the application's observer mechanism for change detection.

A good example is keeping dimensioning consistent with geometry in the CAD domain (See figure 6.2).

6.3.2 Change detection by comparison

The second possibility to detect changed objects is to compare the current state of an object with the state of the same object after the last update. There are several possibilities to achieve this, some of which are discussed below. There is, however, one prerequisite for this technique to be applied at all, namely that the objects are persistently identified. Persistently identified objects have unique names that are session and system independent. A session is defined as the time between the loading and storing of a model by

an application. A persistently identified object always has the same name when loaded into an application, even if the application runs on a different operating system.

It must be noted that not many engineering applications yet exists that provide for persistent object identification. Most current systems will only ensure unique object identification during a single session.

Contents

Change detection may be based on the actual content of an object. In this case the binding mechanism has to log the contents of all the binding objects it contains. In essence it has to create a copy of each and every binding object. When the user initiates an update, the binding mechanism compares the actual state of every binding object with the values stored. If a difference is detected, the object was changed since the last update.

Checksum

The actual state of an object can be expressed in terms of a checksum. A checksum is a long integer that is based on the contents of a byte stream. Thus, if an object is serialized into a byte stream, a checksum-value can be calculated for the serialized object. If any value inside the object changes, the checksum calculation will yield a different value. Consequently, if the binding layer stores the checksum of each binding object and recalculates the checksum at update time, changed objects can be identified. The use of checksums allows change detection on the basis of object contents without the duplication of the actual data contained inside the objects.

The main disadvantage of using checksums for change detection is that it is sensitive to the numerical representation of real numbers. For example, an attribute with a zero value and the same attribute with the value of 1.435E-17 yield different checksums although both values are zero from an application point of view. These small numerical variations have to be accounted for when using checksums for change detection.

Another disadvantage is that different object states may yield the same checksum, although this is highly unlikely. However, changes are not detected if this is the case.

Timestamps

Change detection may be based on timestamps, where each object stores the time when it was last modified. The disadvantage of this approach is that class definitions have to be altered to accommodate the storing of a timestamp, or a listener system has to be introduced to store the timestamps externally. The binding mechanism can then compare the timestamps of the binding objects with the timestamp of the last update event. In the case

that an object's timestamp is later (greater) than its last update event's timestamp, the object was changed and should take part in the next update. This method is hardware dependent, e.g. if the time of a computer is incorrectly set the correctness of the changed set cannot be guaranteed.

Versions

Working with versioned objects [Firmenich 2002; Firmenich et al 2005] allows the monitoring of version numbers to determine if an object was changed or not. Version numbers are not time dependent, thus if the hardware time is incorrect, the changed objects will still be identified correctly.

Versioned objects have the advantage that it is possible to address a specific version of an object. In the construction industry it is useful and necessary to track the development of a project. This is only possible if the information of the project is versioned. Then a specific state of the project is retrievable via the version number of the state.

Based on the discussion above, it is clear that the use of objects with version numbers is superior to the use of timestamps or checksums to detect changes. It is more robust than the other methods discussed and the user remains in control because the versioning decision resides at the user.

Inter-model binders are also denoted as *version-based* binders in this dissertation, because they detect changes based on changed version numbers.

6.4 Updating

Updating is the process of re-establishing consistency between two related models. This process cannot be fully automated since engineering decisions may be necessary to steer and control the effect of changes. Consequently the function of the update process is to *support* the modification of a dependent model to reflect the changes that occurred in the source model since the last consistent state between these two models existed.

The update process has three phases:

- Identification of the changed binding objects and their related bound objects: The changed binding objects are identified as described in section 6.3 above using an observer mechanism for intra-model and version numbers for inter-model dependencies. Their related bound objects, called the affected objects, are immediately available from the unified binding relation described in section 6.1.3.
- Determining the update sequence of the affected objects: The unified binding relation also governs the update sequence of the affected objects. The mathematics and implementation of computing the update sequence is described in section 6.5 below.

- Executing an update of the affected objects: The update mechanism forms the link between the graph that models the object dependencies and the functionality to update the affected objects. A suitable software design pattern for the update mechanism is described below. The main advantage of the pattern is that the update functionality can be assigned at execution time. Furthermore the class definitions of the affected objects do not have to be changed to explicitly support updating, thus a clear separation between the updating mechanism and the middleware model is maintained.

In [Pahl and Beucke 2000; Perevalova and Pahl 2004] a goal set is defined that contains all the objects that must be consistent after the update. This approach reduces the update domain. Objects that are dependent on changed objects are not guaranteed to be updated during the update process unless they are part of the goal set. In this dissertation the goal set is the complete model i.e. the complete model must be consistent after an update and not only a part of it.

6.4.1 Update mechanism

The *binding graph* i.e. the graph that contains all the dependencies between objects, represents the unified binding relation as a set of `Edge` objects[1]. This graph is determined based on the `Binder` instances contained inside the model before an update commences. The update mechanism uses the binding graph to calculate the update domain, i.e. which objects need to be updated. The update domain is also expressed as a graph, i.e. the *update graph*. This graph is topologically sorted to determine the update sequence before the update itself starts. At its core the update mechanism requires two classes, namely the `Binder` and the `Updater`.

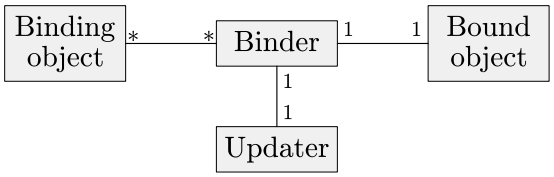


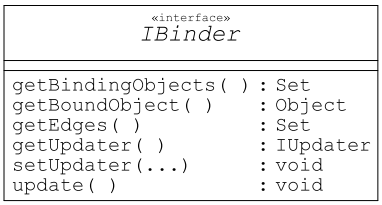
Fig. 6.9: Update mechanism

[1] See section 6.2.3 for mapping a graph to a set of edge objects

Binder

Since some bound objects have dependencies that require more than one binding object, a `Binder` instance is used to group together all the `Edge`s that describe such a dependency. For example, a floor slab may be dependent on a number of lines for its geometry in plan. Consequently its line-to-slab bindings are grouped together in a `Binder` instance.

This ensures that a particular update procedure is executed only once for any affected object, independent of the number of changed binding objects involved in the update.



Allowing the assignment of a bound object to more than one `Binder` instance is possible, provided that the update procedures associated with each `Binder` are independent of one another. For example, the floor slab[2] may also be bound to another object for its thickness. Then two independent updates can be performed when the binding objects are changed. The flexibility of assigning a bound object to more than one `Binder` instance introduces the possibility to model dependencies in an inconsistent way e.g. the thickness of a slab can be bound accidentally to two different objects and, because the update sequence in a specific update step is random, the thickness of the slab may change from update to update due to the specification error.

The actual update functionality is separated from the `Binder` and resides in the `Updater`, to which the `Binder` holds a reference. The need for the assignment of more than one `Binder` instance to a specific bound object can be overcome by using specialized `Updater`s that manage all the dependencies for a specific type of bound object.

Updater

An `Updater` instance provides the functionality required to execute an update on a single affected object. It receives the references of all the objects involved in the update from the `Binder`. The `Updater` is assigned by the user during the process of defining a dependency, which allows the flexibility to switch update functionality at execution time. If no existing `Updater` is suitable, an `Updater` can even be added to the application during execution.

The geometry, design and rebar example introduced earlier is shown in

[2] Assuming that the geometry of the slab is modelled by an arbitrary 2D shape(s) and a thickness, thus the geometric description of the slab is $2\frac{1}{2}$D

figure 6.10. Two `Binder` and two `Updater` instances are required to update the system. The first `Binder/Updater` updates the design to be compatible with the new geometry. The second `Binder/Updater` updates the rebar to be compatible with both the new geometry and design.

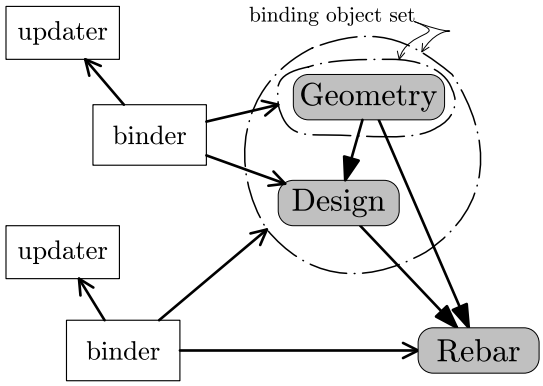


Fig. 6.10: Binders and Updaters

In the absence of `Binder`s and `Updater`s, the following problems will occur:

- All objects that are part of the binding graph will have to implement an update interface that allows the object to be updated. Thus existing classes must be altered to support the binding mechanism.
- The update functionality will reside in the vertices of the dependency graph and cannot be changed dynamically. The class definitions must be altered to change the update behaviour.
- The clear separation between the binding mechanism and the model will disappear.

6.5 Determining the update sequence

The determination of an update sequence is described by [Pahl and Beucke 2000]. An alternative method of determining the update sequence is presented below. The performance of the two methods is compared and the approach presented here displays better behaviour for larger models.

Figure 6.11 presents a binding relation graphically as a graph and mathematically as an adjacency (boolean) matrix. Each vertex of this relation is an object that is referenced by the binding mechanism and each edge represents a dependency relationship. The mathematical theory is explained using this example.

Assume that (b) was changed and the user wants to update the system. The first step is to determine which objects are part of the update domain,

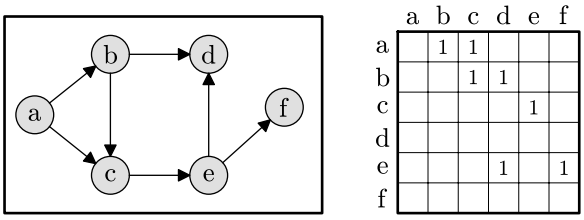


Fig. 6.11: Sorting example

i.e. which objects have to be updated. The second step is to determine the sequence in which the objects are to be updated.

6.5.1 Calculating the update domain

Given the unified binding graph B

$$B := (\Omega; A) \tag{6.9}$$

with vertex set Ω

$$\Omega := \{a \mid a \text{ is a vertex } \} \tag{6.10}$$

and edge set A

$$A := \{(a, b) \in \Omega \times \Omega \mid a \text{ binds } b\} \subseteq \Omega \times \Omega \tag{6.11}$$

the set of changed objects, denoted by P, is determined as described in section 6.3.

$$P := \{a \in \Omega \mid a \text{ was changed } \} \tag{6.12}$$

In general the set P may contain not only binding objects, but bound objects as well. Consider the example of a dimension object that is bound to a line object, i.e. an associative dimensioning object. A user may change only the line, only the dimension, or both. If only the line is changed, the changed object is a binding object. In the case where only the dimension is changed, the changed object is a bound object. When both the line and the dimension are changed, the change-set contains both a binding and a bound object. Binding objects that are changed do not require any updating if they are not bound to other objects. However, bound objects that are changed require updating. In the case of the dimension example, if the dimension end points are changed by the user, the updating mechanism must compare the actual dimension end points with the end points of the line. If these end points are different, the dimension must be changed to remain consistent, or the dependency of the dimension object on the line object must be removed by the user. As a result the set of changed objects is decomposed into two

sets, namely set P_s which contains changed objects that are not bound, and set P_d which contains changed objects that are bound.

$$P_s := \{a \in P \mid a \text{ is not bound } \} \tag{6.13}$$
$$P_d := \{a \in P \mid a \text{ is bound } \} \tag{6.14}$$

where

$$P := P_s \cup P_d \text{ and } P_s \cap P_d = \emptyset$$

The elements of the update domain are determined as described below:

- To start out, the update domain, denoted as set C, is taken as set P.

$$C := P \tag{6.15}$$

- Now, set D is formed as the set that contains all the bound objects contained in set Ω if the respective binding object is contained in set C but the bound object not:

$$D := \{b \in \Omega \mid \bigvee_{a \in C} (a, b) \in A \wedge b \notin C\} \tag{6.16}$$

- The elements of set D are added to set C:

$$C := C \cup D \tag{6.17}$$

- Set D is again defined as in equation 6.16 based on the modified set C, and its elements added to set C. These two steps are repeated until $D = \emptyset$.
- The elements of set P_s, i.e. objects that have originally been changed and are not themselves bound objects, are removed from set C:

$$C := C - P_s \tag{6.18}$$

- Set C now represents the update domain.

Example: Update domain calculation

In figure 6.11 a graph was presented as example. This graph is shown in the page margin again to assist the reader in understanding the example. The update domain calculation is performed based on a changed (b).

Set A is represented below:

$$A = \{(a, b), (a, c), (b, c), (b, d), (c, e), (e, d), (e, f)\}$$

with $C := P$ at the start

$$C = P = \{b\}$$

This example requires four iteration steps to determine all the objects that depend on (b). The sets D and C are shown below after each iteration step.

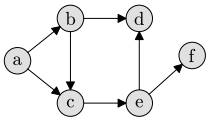
Step 1: $D_1 = \{c, d\}$ and $C_1 = \{b, c, d\}$

Step 2: $D_2 = \{e\}$ and $C_2 = \{b, c, d, e\}$

Step 3: $D_3 = \{f\}$ and $C_3 = \{b, c, d, e, f\}$

Step 4: $D_4 = \emptyset$ and $C_4 = \{b, c, d, e, f\}$

Binding Graph



Set D is now empty, that implies that all the vertices that depend on (b) are now found and that set C contains the update domain.

The investigation of set A indicates that (b) is a bound object, since $(a, b) \in A$. Consequently (b) remains an element of the update domain in order to check that its dependency on (a) is not violated by the change to (b).

If this is not the case, i.e. $(a, b) \notin A$, then the edges (b, c) and (b, d) are removed from the update domain in order to remove (b) from the update graph.

6.5.2 Topological sorting

All the elements of the update domain are known after the calculation described in the previous section. The correct sequence of updating these elements requires that an object cannot be updated before all objects on which it depends have been updated. In graph theory terms this means that all predecessors of a vertex have to be updated before the vertex itself can be updated. This is a known problem in graph theory and the solution is found by performing a topological sorting of the graph [Pahl and Damrath 2001]. Topological sorting assigns a step number to each vertex of the graph. The step number indicates the earliest opportunity in the update sequence when the vertex can be updated, with the assurance that all its predecessors have then already been updated.

Define the update graph as:

$$G := (N; L) \tag{6.19}$$

with vertex set N

$$N := \{a \mid a \in C\} \subseteq \Omega \tag{6.20}$$

and edge set L

$$L := \{(a, b) \in N \times \ N \mid a \text{ binds } b\} \tag{6.21}$$

The first step for the sorting algorithm is to define a set S, that contains all the vertices of the update domain i.e. all the vertices of the update graph. As the sorting algorithm continues, vertices will be removed from this set and added to the update sequence until S is empty or a cycle is found. If a cycle is found it is not possible to calculate the update order directly and an intermediate step is required as described in the next section.

$$S := N \tag{6.22}$$

The topological sorting of the graph is executed inside a loop. Each time the loop starts again indicates a new step inside the update order. The loop executes until a cycle is found or all vertices have been assigned to their respective steps.

Loop start

- Define set E that contains all the end vertices of the ordered pairs inside L. Consequently set E contains all the objects that are still dependent on other objects inside the relation.

$$E := \{b \in S \mid \bigvee_{a \in S} (a, b) \in L\} \tag{6.23}$$

- Set U contains the difference between S and E. The elements of set U do not depend on other objects inside the relation and can be updated in the current step. The update order of these objects amongst themselves is irrelevant because they do not depend on one another.

$$U := S - E \tag{6.24}$$

- Set L is modified by removing the ordered pairs whose start vertices are elements of U, i.e. they form part of this update step.

$$L := L - \{(a, b) \in L \mid a \in U\} \tag{6.25}$$

- Set S is also modified by removing all the vertices that were updated.

$$S := S - U \tag{6.26}$$

Operations defined in equations 6.23 to 6.26 are repeated until $U = \emptyset$ or $S = \emptyset$

Loop end

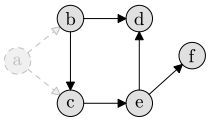
Cycles

When the loop terminates and set S is not empty it implies that there exist dependencies between some of the remaining ordered pairs that form a cycle inside the dependency graph. A cycle may indicate an error in the dependency specification, which can be corrected before updating is attempted again. If a cycle occurs and there is no error in the dependency specification, replacing the vertices that are part of the cycle with a single macro vertex allows the sorting algorithm to continue [Pahl and Beucke 2000]. After the sorting algorithm is finished, the macro vertex is replaced by the vertices that it represented during the sorting algorithm.

Example: Topological sort calculation

Continuing with the example introduced at the start of this section, the update order now needs to be calculated.

Update Graph



This example starts of with

$$L := \{(b, c), (b, d), (c, e), (e, d), (e, f)\}$$

and

$$S := \{b, c, d, e, f\}$$

Again this example requires four iteration steps to determine the update sequence. The sets L, S, E and U are shown below after each iteration.

Step 1: $E_1 = \{c, d, e, f\}$, $U_1 = \{b\}$
set L and S change to $L_1 = \{(c, e), (e, d), (e, f)\}$, $S_1 = \{c, d, e, f\}$

Step 2: $E_2 = \{d, e, f\}$, $U_2 = \{c\}$
set L and S change to $L_2 = \{(e, d), (e, f)\}$, $S_2 = \{d, e, f\}$

Step 3: $E_3 = \{d, f\}$, $U_3 = \{e\}$
set L and S change to $L_3 = \emptyset$, $S_3 = \{d, f\}$

Step 4: $E_4 = \emptyset$, $U_4 = \{d, f\}$
set L and S change to $L_4 = \emptyset$, $S_4 = \emptyset$

Update Sequence

1. b
2. c
3. e
4. d f

After step 4 the topological sort algorithm teminates because set L is empty. This implies that the update sequence was successfully calculated.

Matrix representation

A boolean matrix representation of an update relation shows the relationships between the individual objects. Rows represent outgoing edges from a vertex and columns represent incoming edges to a vertex. An entry in the boolean matrix contains a *one* (true) if there exists an edge from the row's vertex to the column's vertex otherwise it contains a *zero* (false). When visualizing a boolean matrix, the *zero*s are omitted to make the matrix more readable.

The example presented in figure 6.11 is now used to indicate the changes to the adjacency matrix as the sorting algorithm progresses. Figure 6.12 shows the changes to the adjacency matrix after each iteration of the sorting algorithm.

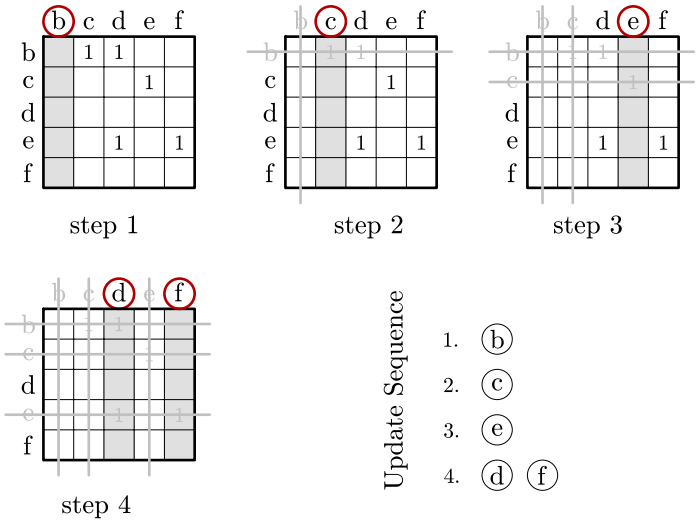


Fig. 6.12: Example 2: Sorting process, Matrix view

The column of vertex b is empty, which implies that there are no incoming edges to b. Thus, b depends on no other vertex in the relation and should be updated next (update step 1). Once the update is performed, b as well as all the outgoing edges from b are removed from the update graph. This ensures that the update graph only contains objects that are not up to date yet and reduces the size of the update graph by removing the updated objects. The removal of b and its outgoing edges from the update graph is represented in the boolean matrix by crossing out the row and column that represents b (row 1 and column 1 are crossed out).

The matrix is now re-evaluated. The column of vertex c is now empty, which implies that there are no incoming edges to c, thus c should be updated next (update step 2). This process of searching empty columns and removing empty rows and columns is repeated until all the vertices are updated. After

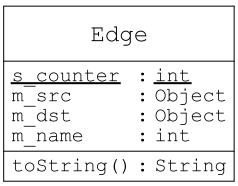
each update step, the size of the update relation is reduced and one or more rows and columns are crossed out. In step 4, both d and f contain no more incoming edges, thus they can be updated in the same step.

If a situation arises where there are no empty columns in the boolean matrix after an update step, a cycle exists in the update graph which should be dealt with as described before.

6.5.3 Object-oriented implementation

The ordered pairs (a, b) of the binding relation are represented by `Edge` objects. An `Edge` object has a reference on the source object a and the destination object b of the ordered pair (a, b) that it represents.

The `Edge` object instances are transient in the sense that they are created by the `Binder` objects that are contained inside the binding mechanism just prior to an update and are discarded after the update.



The sorting algorithm is implemented using Java. The first step is to determine the update domain. The source code for this is presented in listing 6.1. Afterwards the update domain is sorted topologically.

Update domain implementation

```
public static Set<Edge> getUpdateDomain(Set<Object> changed,
    Set<Edge> relation){
  Set<Edge> _relation = new HashSet<Edge>(relation);
  Set<Edge> _affected = new HashSet<Edge>();
  List<Object> list = new ArrayList<Object>(changed);
  while(list.size() > 0){
    Object obj = list.remove(0); // use list as queue (FIFO)
    Iterator<Edge> iter = _relation.iterator();
    while(iter.hasNext()){
      Edge e = iter.next();
      if(e.m_src.equals(obj)){
        _affected.add(e);
        list.add(e.m_dst);
        iter.remove();
      }
    }
  }
  // determine vertices that are not bound, but
  // part of change set.
  Set<Object> bindingObject = new HashSet<Object>(changed);
  Iterator<Object> vertex_iter = bindingObject.iterator();
  while(vertex_iter.hasNext()){
    Object vertex = vertex_iter.next();
    Iterator<Edge> edge_iter = relation.iterator();
    while(edge_iter.hasNext()){
      Edge e = edge_iter.next();
      if(e.m_dst.equals(vertex)){
        vertex_iter.remove(); // implies vertex is bound
        break;
      }
    }
  }
  // remove edges where binding object is not bound
  vertex_iter = bindingObject.iterator();
  while(vertex_iter.hasNext()){
    Object obj = vertex_iter.next();
    Iterator<Edge> edge_iter = _affected.iterator();
    while(edge_iter.hasNext()){
      Edge e = edge_iter.next();
```

```
40          if(e.m_src.equals(obj))
              edge_iter.remove();
          }
        }
        return _affected;
45    }
```

Listing 6.1: Determination of the update domain

The `getUpdateDomain` method receives two sets: The first set, *changed*, contains all the vertices that were changed. The second set, *relation*, contains all the `Edges` that are part of the binding mechanism. The *relation* set is copied into a new set *_relation*. This is done because the algorithm removes objects from the second set during the search and it should not change the set handed to it. The method returns the set, *_affected*, that is computed as described in section 6.5.1. Internally, this method uses a list, *list*, that contains all the objects contained in *changed* at the start. The following two steps are repeated inside a loop until *list* is empty.

- the object at index zero is removed from *list* and referenced by variable *obj*.
- Iterating over all the `Edge` objects inside *_relation*:

  If an `Edge` *e* is found where the source object (i.e. 'first' vertex, m_src) of *e* is the same object as *obj*, then the destination object (i.e. 'second' vertex, m_dst) of *e* is dependent on *obj*. If this is the case the following actions are taken:

  1. *e* is added to set *_affected*.
  2. The destination object of *e* is added to *list*
  3. *e* is removed from *_relation*.

  At a later step in time the destination object of *e* will be removed from *list* and the edges that depend on it will be searched.

After all the affected objects have been determined, the vertices that are not bound objects but are contained in *changed* are identified, see listing 6.1 lines 18-32. The corresponding edges are then removed from *_affected*, see listing 6.1 lines 33-43. The set *_affected*, which represents the update graph, is returned.

Topological sort implementation

The algorithm for performing topological sorting of the update domain was explained in section 6.5.2. The Java implementation is shown in listing 6.2.

```
public static List<List<Object>> topologicalSort(
    Set<Edge> updateDomain){
  List<List<Object>> list = new ArrayList<List<Object>>();
  if(updateDomain.isEmpty())
5   return list;
  int _counter = 0;

  Set<Object> start = new HashSet<Object>();
```

```
    for(Edge e: updateDomain){
      start.add(e.m_src);
      start.add(e.m_dst);
    }
    Set<Object> completed = new HashSet<Object>();
    while(true){
      Set<Object> end = new HashSet<Object>();
      Iterator<Edge> it = updateDomain.iterator();
      while(it.hasNext()){ // populate E
        Edge e = it.next();
        if(!completed.contains(e.m_src))
          end.add(e.m_dst);
        else
          it.remove();
      }
      list.add(new ArrayList<Object>());
      Iterator iter = start.iterator();
      boolean cycle = true;
      while(iter.hasNext()){ // S - E
        Object obj = iter.next();
        if(!end.contains(obj)){
          cycle = false;
          list.get(_counter).add(obj);
          completed.add(obj);
          iter.remove();
        }
      }
      if(cycle)
        throw new RuntimeException(
            "Cycle Detected!!!\n"+list);
      _counter++;
      if(end.isEmpty())
        break;
    }
    return list;
  }
```

Listing 6.2: Topological Sort Implementation

The sort algorithm receives a set of `Edge` objects. This set contains all the ordered pairs (a, b) of the update domain as determined in the previous algorithm, see listing 6.1. Firstly, all the vertices of the `Edge` set are added to a set called *start*. Another set called *completed* is created. This set contains all the vertices that were removed from the update domain, i.e. the vertices that have been sorted. A while loop is started that terminates when all vertices have been sorted or when a cycle is found. The following steps are repeated inside this loop.

- a set *end* is created.
- Iterating over all the `Edge` objects inside *updateDomain*: If an `Edge` is found where the source ('first') object of the `Edge` is not contained inside *completed*, the destination ('second') object of this `Edge` is added to the *end* set (Equation 6.23).
- Iterating over the *start*: If an object is found that is not contained inside *end*, it implies that there are no more incoming edges into this object (i.e. the column of the boolean matrix of this vertex is empty). This vertex is now added to *completed*. It is also added to the update list (Equation 6.24) and removed from *end*. The boolean flag that indicates the existence of cycle is set to false.
- If no object was removed during the previous step, the cycle flag will be true and a RuntimeException is thrown to indicate the existence of a cycle.

Testing the sorting algorithms

The example in figure 6.11 is sorted using the algorithm presented in this section (denoted as AHO), and the algorithm documented by [Pahl and Beucke 2000] (denoted as IKM). The class that implements the example is shown in listing 6.3. The output of this example is presented afterwards in output 1.

```
package thesis;
import java.util.*;
import thesis.utils.Out;

public class Example {

  public static void main(String[] args) {
    // create six objects
    Object[] vertex = new String[]{"a","b","c","d",
        "e","f"};

    //create edges
    Set<Edge> relation = new HashSet<Edge>();
    relation.add(new Edge(vertex[0],vertex[1])); // a -> b
    relation.add(new Edge(vertex[0],vertex[2])); // a -> c
    relation.add(new Edge(vertex[1],vertex[2])); // b -> c
    relation.add(new Edge(vertex[1],vertex[3])); // b -> d
    relation.add(new Edge(vertex[2],vertex[4])); // c -> e
    relation.add(new Edge(vertex[4],vertex[3])); // e -> d
    relation.add(new Edge(vertex[4],vertex[5])); // e -> f

    Set<Object> changedVertices = new HashSet<Object>();
    changedVertices.add("b");

    Set<Edge> changeSet = new HashSet<Edge>();
    for(Edge e : relation){
      if(changedVertices.contains(e.m_src))
        changeSet.add(e);
    }

    Set<Edge> affectedSet =
      TopologicalSort.getUpdateDomain(changeSet, relation);

    List<List<Object>> sorted =
      TopologicalSort.topologicalSort(affectedSet);
    System.out.println("AHO RESULTS:");
    Out.printList("update order", sorted);

    System.out.println("\nIKM RESULTS:");
    Graph g = new Graph(relation);
    boolean[] changedObj
      = g.getChangedObject(changedVertices);
    boolean[][] aMat = g.aMat;
    boolean[] updateDomain
      = Ikm2000.updateDomain(changedObj,g.aMat);
    boolean[][] reducedAMat
      = Ikm2000.chronoMatrix(updateDomain, aMat);

    int[] age = Ikm2000.computeAge(reducedAMat);

    Out.printVec("Age vector", age);
    Out.printList("update order",
        g.getUpdateSequence(updateDomain, age));
  }

}
```

Listing 6.3: Topological Sort Example

Output 1 firstly shows the results of the presented algorithm and then the results of the algorithm described in [Pahl and Beucke 2000]. The results are identical although the underlying methodologies are different.

Output 1 Topological sort example

```
AHO RESULTS:
update order :
 step 0 : b,
 step 1 : c,
 step 2 : e,
 step 3 : f, d,

IKM RESULTS:
Age vector:
0 1 3 2 3
update order :
 step 0 : b,
 step 1 : c,
 step 2 : e,
 step 3 : d, f,
```

Performance of implementation Comparing the performance of this implementation to the implementation approach of [Pahl and Beucke 2000] yields the following results.

- for small update graphs the IKM algorithm performs faster.
- sorting medium to large graphs, the AHO algorithm performs faster.
- IKM algorithm is more sensitive for number of vertices.
- AHO algorithm is more sensitive for number of edges.
- IKM algorithm will fail before the AHO algorithm.

See appendix C for a summary of the tests that were performed in comparing these two algorithms.

6.6 Performing the actual update

The vertices in the update graph are the objects that need to be updated in order to restore consistency in the model. The topological sort algorithm, explained in section 6.5.2, structures the vertices of the update graph into the update sequence. Each vertex is a bound object, thus each vertex has an assosiated `Binder` instance. This allows the mapping of the update sequence onto the set of `Binder` instances in order to structure this set in such a way that before the `update()` method of a specific `Binder` instance is invoked the updates of all the preceding vertices are completed.

6.7 Summary of update procedure

Figure 6.13 presents a visual summary of the proposed update mechanism.

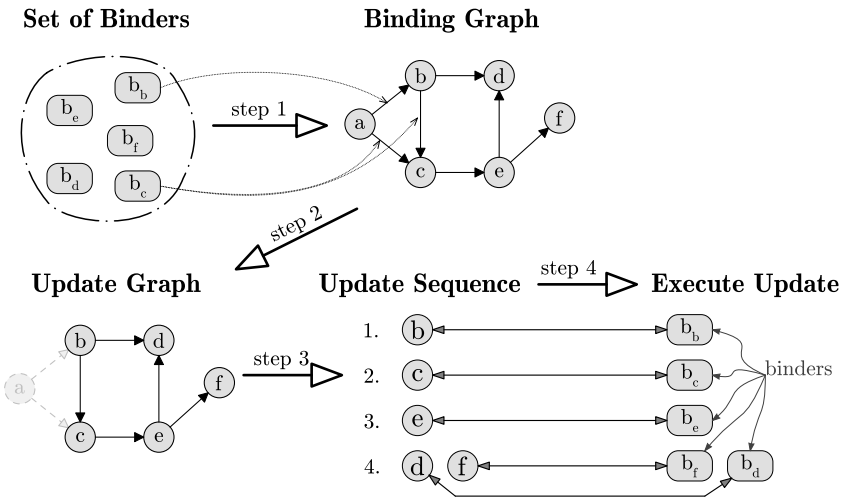


Fig. 6.13: Overview of update process

The inter- and intra model dependencies are captured inside `Binder` instances. These `Binder` instances reside inside a data structure that provides access to it. In figure 6.13 the `Binder` instances are contained in a set denoted as 'Set of Binders'.

The following steps occur when a user performs an update:

1. The `Binder` instances are traversed. Each `Binder` instance returns one or more `Edge` objects. Each `Edge` object represents a dependency between two objects. These edges form the *binding graph*.
2. The update domain is determined, see section 6.5.1, based on the objects that were changed since the last update. The result of this step is the *update graph* which is represented as a set of edges.
3. The update sequence is determined, see section 6.5.2, by performing a topological sorting of the update graph.
4. The update sequence is used to structure the set of binders and then the `update()` method of each `Binder` instance is invoked.

Note that in this example the update method of (b) is invoked to ensure that the changes made to (b) do not violate the defined dependency between (a) and (b). The result may be that (b) is restored to the same state as before the changes occured.

Chapter 7

Working with Structural Engineering Models

The Structural Engineering Model (SEM) was introduced in chapter 4 and extended in chapter 5.

The purpose of a SEM application is to support a structural engineer in defining the structural concept, i.e. the loadbearing system of a building using existing CAD information and CAD tools. The resulting computer model, i.e. the SEM, represents the structural components of the building under consideration. These components encapsulate the structural properties and semantics provided by the engineer and contain all the information required to create a FEM instance of the structural concept. The SEM components are bound to CAD information for the purpose of supporting consistency in case architectural details of the building change.

The SEM application is envisaged as an important tool in the structural analysis and design stage of a building since it has to replace the "drawing board" that the structural engineer typically uses to define the structural concept of a building. The change may be compared to the step that structural draftsmen took some years ago when they started doing their daily work on a computer rather than on a paper and pencil drawing board.

This chapter explains in detail what an end-user could expect from an application that operates on such a model. The last part of this chapter presents the pilot application, called structures@CADEMIA, that was developed as proof of concept.

7.1 Standalone SEM applications

Currently, commercial CAD systems [Autodesk 2007] allow the creation of building components rather than 2D lines or 3D solids. The SEM application is an extension to such a product, allowing the structural engineer to specify aspects of the building that only concern structural analysis and design. In

the process he may use information contained in an existing CAD model of the building, in which case support is provided to keep the information in the CAD and SEM models consistent. However, the CAD and SEM models are completely seperate. Alternatively a SEM model can be created without an underlying CAD model.

The behaviour of a SEM application is similar to an ordinary CAD system, allowing a user to interact with a model using CAD tools. Standard CAD operations can be decomposed into three categories namely addition, modification and removal of components. In the SEM application domain, these operations are performed likewise:

add a component with engineering semantics, i.e. a slab or a load. An engineering component usually comprises geometry, e.g. outline of slab and/or properties, e.g. material type of slab.

modify the geometry and/or properties of an engineering component. The outline of a slab (geometric change), or the material properties of a slab (property change) may be modified.

remove a component from the model.

In addition to the above, standard CAD functionality enables the user to select, copy, move, rotate, scale and mirror SEM components. The properties of the components are also modifiable via the application interface. If a user is familiar with a normal CAD system and has the necessary structural analysis background, the creation and manipulation of a SEM instance is intuitive and it is not foreseen that additional training is required to make use of a SEM application.

7.2 SEM components

The components of a SEM instance are divided into three categories, discussed in the sections below. In line with the research focus of the dissertation, no attempt is made to establish an exhaustive list of components. However, the pattern for component definition is established sufficiently.

7.2.1 Physical building components

These components comprise physical building elements constructed to shape the building. They also have meaning in other disciplines. Typical examples are columns, load-bearing walls, shafts, foundations, trusses, etc. Their function inside the SEM is to represent the physical layout of the load bearing system. Two physical components were implemented in the pilot application:

Slab: This component represents a floor slab. It contains the following additional information:

- thickness
- material type

Beam: This component represents a reinforced concrete beam that is cast monolithically with the slab. It could either be an upstand or downstand beam. It contains the cross sectional properties of the beam, including the offset of the center of the beam to the center of the slab and the material type.

7.2.2 Numerical components

Numerical components only exist in the structural analysis and design domain. They are used to model structural behaviour and to express abstract aspects of the design intent of the structural engineer. Examples of numerical components are loads, supports, local coordinate systems and constraint conditions.

The following numerical components were implemented in the pilot application:

loads acting on the slab:

- point load.
- line load.
- area load.

supports that suspend the slab:

- point support.
- line support.

material properties that define Young's modulus, Poisson's ratio and the density of the slab and beam materials.

cross sectional properties that define the moments of inertia of the beam components.

7.2.3 Geometry independent components

Some components of the SEM are geometry independent. The following geometric independent components were implemented in the pilot application:

- a loadset that groups load instances together into a single load case applied to the structure.
- meshing parameters that are used by the meshing algorithm to control the resulting mesh, e.g. number of elements in the mesh or the average element size.

The creation and modification of these components do not require operations on geometric entities.

7.2.4 Mapping geometry to SEM components

If the SEM model is based on an underlying CAD model, entities in the CAD model are mapped to entities in the SEM model by the structural engineer. This mapping depends on the structural concept of the engineer, and cannot be automated. For example, a column in the geometric description may be mapped to a column in the SEM, to a point load on a slab or to a point support of a slab, depending on the concept of the engineer. Software cannot make this decision independently based on rules like *'if the column is under the slab it is a support and if the column is on top of the slab it is a load'*, because the role of the column is determined by the structural engineer. If the column is used as a hanger, i.e. it works in tension to support a slab from above, it actually acts as a support and not as a load on the slab.

7.3 Creation of SEM intances

The creation of geometry dependent SEM components can be achieved in two ways. One approach is to select existing geometric entities and use that to create a SEM component, the other is to create SEM components directly.

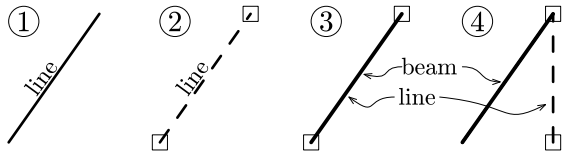


Fig. 7.1: *Select-and-create* approach to instantiate SEM instances

Figure 7.1 presents an example of a beam instance created by the *select-and-create* approach, which entails the following steps:

1. a line is created that represents the geometry of the beam
2. the line is selected
3. a beam is created on top of the selected line
4. the bottom point of the line is moved, but the beam remains in the same position, i.e. the geometry of the line was tranferred by value into the beam and there exists no connection between the beam and the line component after the beam was created.

Another way of defining a SEM component is to define the SEM component from scratch. In terms of the beam example above, the beam is constructed in the same way as the line, with the only difference that the end result is only a beam component and not a line.

Both beam instances are exactly the same although the way in which they have been created is fundamentally different.

The advantage of the *select-and-create* approach is that a single command can create SEM instances based on different source geometry types, e.g. lines, rectangles and bezier curves. From an application development point of view this implies that a single additional command can create components with different geometry types. Furthermore the *select-and-create* approach is in line with the concept that the SEM model is defined using geometry available in an existing CAD model. The disadvantage is that a user might perceive the two steps as more effort than just 'drawing' the beam instance directly, similar to drawing a line.

The two ways of defining SEM components do not exclude one another, and both can be implemented in the same SEM application.

In the pilot application a *select-and-create* approach to define the geometery of a SEM component was implemented.

7.4 Deriving the FEM

Once the structural engineer is satisfied that the SEM instance represents the structural concept under consideration, a FEM instance of the building can be derived. The FEM instance is analysed and the results are then evaluated by the structural engineer. If the structural engineer is satisfied with the results, the design of the structural components continues.

Before the actual derivation process starts, the SEM instance is validated. The validation step, for example, may check that all the loads and supports act on the slab. Different validation algorithms can be implemented to ensure that the quality of the SEM is acceptable before the FEM instance is derived.

The functionality to derive a FEM instance from a SEM instance does not form part of the SEM model itself. It is situated outside the model, which allows a SEM instance to feed different FEM applications. This makes the use of a SEM application attractive because it is independent of the actual finite element application that is used. The pilot application uses a text file as an interface between the SEM and the finite element application. Thus, by changing the format of the text file, different applications can be used to perform the finite element analysis.

For the slab component implemented in the pilot application, the main task during the derivation step is to calculate a suitable finite element mesh. A simple mesher was developed for this task to keep the pilot implementation independent of third party software. The mesher has the following properties:

- It produces a triangular mesh of the slab entity under consideration.
- It provides nodes at point loads and point supports.
- It refines the mesh at point supports.

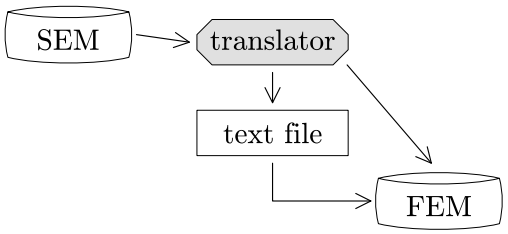


Fig. 7.2: Deriving the FEM instance

- It ensures that line loads and line supports do not run through elements, but always coincide with an element edge.

Once the mesh is created, the supports and loads are added, as derived from the corresponding components of the SEM instance.

7.5 Binding a SEM instance to existing geometry

The solution approach to bind a SEM instance to existing geometry requires a user to specify the dependencies. A `Binder` instance is an object that models the dependency between a set of source objects and a destination object as described in chapter 6.

Two possibilities exist to create a `Binder` instance. The first possibility is to separate the creation of the SEM component from the creation of the `Binder` instance. Thus, a user first creates a SEM component and then specifies its relationship to the geometry by which it is bound.

The second possibility is to pre-select the binding geometry and then issue the command that creates the SEM component (See figure 7.1). This approach of *select-and-create* is useful because the engineer assigns semantics to existing geometry and thus implicitly defines the binding relationship between geometry and SEM components. The main advantage of this approach is that the engineer does not explicitly have to create the binding component that models the dependency relationships between the geometry and the SEM component, it is included in the creation of the SEM component. Furthermore it requires less implementation effort.

The pilot application uses the *select-and-create* approach to create the SEM components and their bindings. Once a SEM component is created, it functions completely independently of the geometry on which it is based in order not to violate the pre-requisite that models must remain separate.

7.5.1 Binder classification

Binders are categorised based on their origin, domain and dependency cardinality.

Origin

A binder is either automatically or manually created by the user. The automatic creation of binder instances has the advantage that the user does not need to take care of binder creation. Manually defining the binder instances provides the user with flexibility to bind any object instance to any other object instance. The two ways of defining binder instances do not exclude one another. Both ways are supported by the pilot application. The only disadvantage of allowing a user to specify a binder instance manually is that cycles can be introduced into the update graph by incorrect user input.

Domain

A binder is either active between objects of a single model (intra-model), or between objects that reside in different models (inter-model).

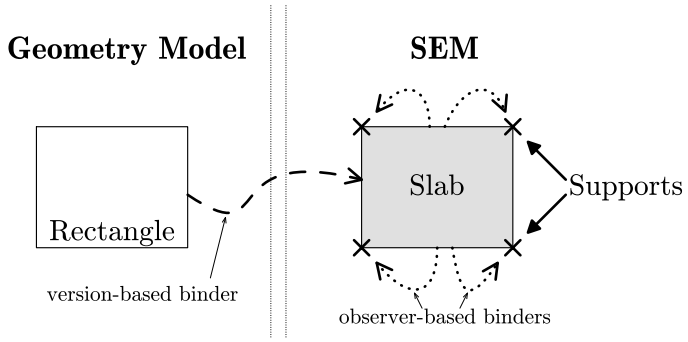


Fig. 7.3: Version-based vs observer-based binders

Figure 7.3 shows two models, on the left-hand-side is a geometry model that contains a rectangle. This rectangle is used as the outline of the slab that is contained in the SEM model on the right-hand-side of the figure. There exists one binder instance between the rectangle and the slab component. This binder is an inter-model binder, i.e. version based[1], because the source and the destination objects are located inside separate models.

This SEM also contains four support objects, one at each corner point of the slab instance. Each of the supports is bound to the corresponding corner of the slab instance[2]. These binder instances are intra-model binder instances, i.e. observer based.

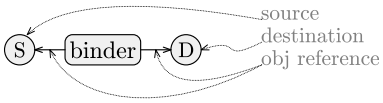
[1] version based implies that change detection is based on the version numbers of the binding objects, see section 6.3.2

[2] A support depends on a specific slab corner i.e. during an update the supports are moved to coincide with the slab corners and the geometry of the slab is not affected by the support locations

Dependency cardinality types

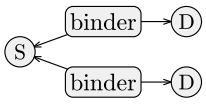
Four types of dependency cardinality between objects are possible. These are one-to-one, one-to-many, many-to-one and many-to-many dependencies. The cardinality types are supported by the creation of one or more `Binder` instances in order to model the inter-object dependencies in the following way:

one-to-one This dependency type is the simplest form of dependency. It is modelled by one binder instance that contains one source and one destination object.



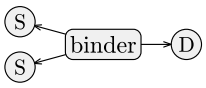
The binder instance between the rectangle and the slab object in figure 7.3 is an example of such a dependency.

one-to-many A one-to-many dependency is modelled as several independent one-to-one dependencies.



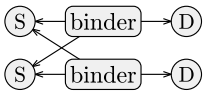
In the example above all four support objects depend on the slab instance. This is modelled by four binder instances.

many-to-one This dependency type is modelled with one binder instance that contains more than one source object and one destination object.



An example is a slab instance that is derived from and bound to several individual geometry instances. If one of the geometry instances changes, the slab must be recalculated as a whole.

many-to-many A many-to-many dependency is modelled as several many-to-one dependencies.



7.5.2 Re-using existing geometry

The SEM application needs to give the user access to existing CAD geometry as the starting point when a new SEM instance is created, i.e. the SEM application needs to import existing information.

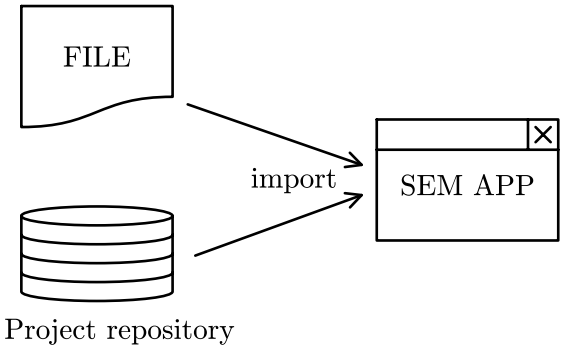


Fig. 7.4: Different geometry sources

The import task is responsible for retrieving information from another system and to make it available to the structural engineer in the creation of the SEM instance, see figure 7.4. Imported geometry must fulfill the following requirements in order to support the binding mechanism:

Persistent identification: [van Rooyen 2002] Any object must be identified by name, i.e. independently from the runtime environment of an application.

Versioning: Objects must be versioned in order to detect changes based on version numbers.

When the structural engineer decides to update a SEM instance, the import command is executed to re-import the source geometry. Changes between the original and the newly imported geometry are detected by comparing the actual version number with the version number of the object that the SEM component is bound to. When a difference is detected, the binding object was changed since the SEM component was created, thus an update is required.

The pilot implementation uses a CADEMIA instance as source to feed the SEM application with persistently identified, versioned geometry. In a commercial application which supports bindings between objects in different models, the source information would typically reside on a project server and the user would send a query over a network to obtain the required information. In order to achieve this, the only modification to the SEM application is to create a new *import* command.

7.6 Classification of changes

Consistency between two models can be compromised if changes occur in either one of the models. The different types of changes that can compromise the consistenty between the two models are described below.

7.6.1 Changes to geometry model

Before an update is performed to ensure consistency based on the parent geometry, a re-import is performed based on the same parameters as the original import event. The following events may have occurred in the geometry model in the time frame between the two import events:

Objects added to geometry model

If new geometry is returned by the re-import command, the new geometry is evaluated in the same way as the originally imported geometry. Thus, the engineer makes decisions to assign structural semantics to the newly found geometry and creates the corresponding binder instances.

Objects removed from geometry model

If geometry was removed from the geometry model between the first import and the re-import event, the corresponding binder instance detects that the geometry that binds a SEM component is not part of the actual geometry model anymore. The structural engineer then needs to make a decision to either remove the SEM component or to keep the SEM component as it is. If he decides to keep the SEM component, the binder instance is removed and the the SEM component remains untouched.

Objects changed inside the geometry model

If changes to the geometry are detected, the update mechanism supports the structural engineer in updating the SEM instance to be consistent with the changed geometry.

On the side of the geometry model a change can be achieved by removing a component and then adding a new component. Although the result of such

an event is the same as transforming the original geometric component into a new state, the system cannot detect this type of change event because the new component has a name and version number that differs from that of the original component. If the structural engineer interprets that a change was performed via *adding* and *removing* geometry, he needs to update the binder instance to use the new geometric instance as binding object before the update process is invoked.

7.6.2 Changes to SEM models

The SEM instance functions completely independently of the geometry model that was used as its original source. The structural engineer is not restricted in any way to perform actions that violate existing bindings. For example, if a beam is bound to a line in the geometry model, the structural engineer is free to move the beam to another location.

The binder that models the dependency may not and should not prohibit the engineer to work freely with the SEM instance. However, before any update step is performed, the engineer has to decide whether the binder is to be modified to bind the new location of the beam to the geometry in such a way that after the update process the beam is still at the new location. In this scenario the beam remains a bound object inside the SEM instance.

Another option is to remove the binder instance from the model so that subsequent updates will not affect the beam.

The third option is no intervention by the structural engineer, in which case the changes made by the engineer are undone by the binder and the beam location is changed back to be consistent with the binding geometry.

7.6.3 Supporting the engineer in preparing the actual update

An important aspect of creating and maintaining dependencies between two or more models is how to communicate the state of the binder instances as well as the state of the binding and bound objects to the owner of the bound model instance.

A binder instance cannot be visualized on its own due to the fact that it only represents a dependency between two or more components. However, allowing a user to select an object via mouse pick and then highlighting the object as well as the binding objects helps the user to identify an individual binding.

An example application was developed to simulate a sequence of events[3] that occur in order to explain how a SEM application functions with specific reference to binder behaviour. Figure 7.5 shows the GUI of the example application. The GUI contains a menu on the left-hand-side that performs predefined actions. These actions are used in the example that follows to

[3]For example, geometry added to geometry model

perform a sequence of events that simulates the engineer/application interaction. The main part of the GUI displays two models, the top model is the imported geometry and the bottom model is the SEM instance that is based on the imported geometry. No CAD functionality is provided to edit either of the models since that falls outside the scope of the example.

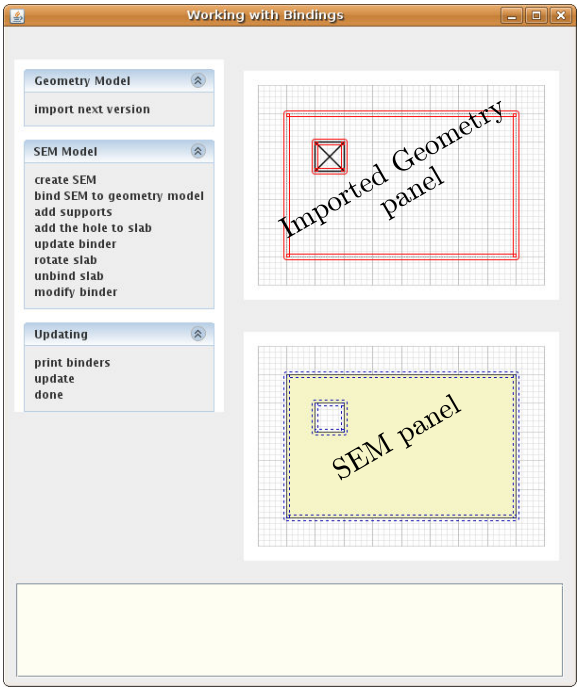


Fig. 7.5: Binding behaviour example

Functionality to indicate which object dependencies exist is implemented. If an object is selected the outline of the object is highlighted with a dotted line and all the objects that bind the selected object are also highlighted with a solid line. In figure 7.5 the slab instance of the SEM model was selected. The geometry that binds the slab and the slab are highlighted accordingly. It was concluded that meaningful support to visualize object dependencies is provided in this way.

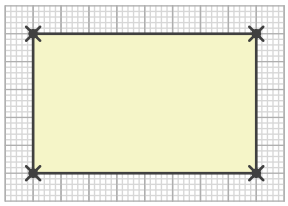
Another way of communicating dependencies is to use different colours and/or line types to show all the bound objects inside the SEM model. This immediately gives the owner an idea which components are bound and which are not bound. Furthermore, highlighting the new and changed geometry after each re-import assists the engineer in identifying areas where he needs to focus his attention. This is of great value, especially if the number of re-imported components is high.

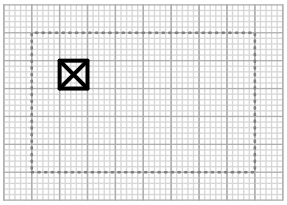
Example: SEM model creation and binding behaviour

The figures on the left-hand-side show the geometry panel after a specific event, likewise the figures on the right-hand-side show the SEM panel after a specific event.

First import The first step is to import the geometry that serves as basis for the SEM instance. The figure shows a panel that contains the geometry model after the initial import. In this example the geometry model comprises only one rectangle in order to keep the complexity of the example to a minimum.

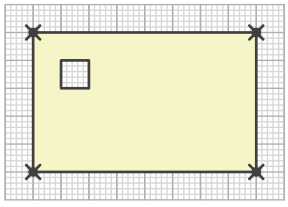
Slab creation In the second step the structural engineer interprets the imported geometry, and creates a slab component based on it. The dimensions of the slab are the same than the dimensions of the imported rectangle. The figure shows a panel that displays the SEM instance after the creation of the slab component. Four supports are also created at the corners of the slab. The binders between the supports and the slab are *observer-based* binders and the binder that binds the slab to the rectangle is a *version-based* binder.

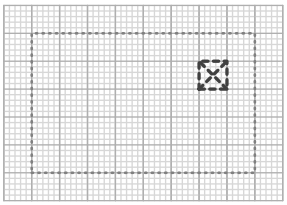




Re-import geometry A re-import of the geometry is performed. The new geometry model is shown in the figure. The geometry that was part of the previous import is represented by gray dotted lines and the geometry that was added since the previous import is displayed by solid black lines. This helps the engineer to assess the newly imported geometry.

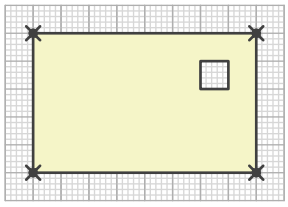
Adding opening to slab The added geometry is interpreted as an opening in the slab. The engineer adds the opening to the slab instance by selecting the appropriate item in the SEM menu. The slab geometry is bound to the original rectangle as well as the newly added opening. The binder is also updated during this step.

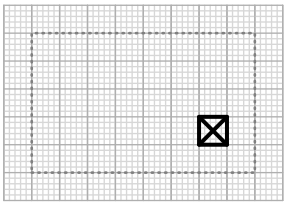




Re-import of geometry Another re-import of the geometry that binds the SEM instance is performed. The new state of the geometry model is shown in the figure. The outline of the slab remained unchanged since the last import, while the position of the opening was moved to a new location. Imported geometry components with new version numbers are displayed with dotted lines. This assists the engineer in identifying changes that occurred in the geometry model.

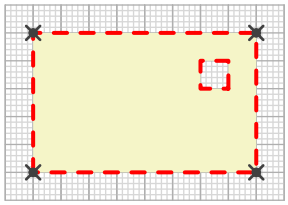
Updating the slab The engineer recognizes that some geometry that binds the slab was changed since the last import and invokes the update mechanism. The update procedure first calculates the update domain and then the updating sequence. It then invokes the updaters associated with the individual objects of the update domain in order to make the SEM instance consistent with the new geometry.



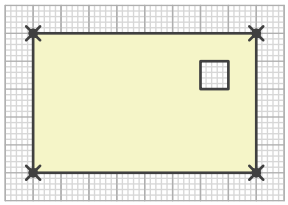


Re-import of geometry Again the geometry is re-imported. The geometry that represented the opening in the slab was removed since the previous import and new geometry was added by the owner of the geometry model. The new geometry is highlighted by solid black lines and the geometry that was removed is not displayed at all.

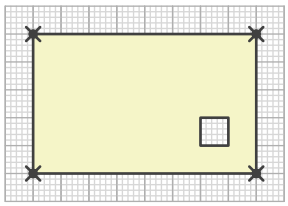
Binder violation The slab instance is bound to geometry that was removed since the last import, therefore it is not possible to perform an immediate update to restore consistency between the geometry model and the SEM instance. The outline of the slab changes to a thick dashed line to indicate that the engineer needs to decide whether the binder is to be removed or updated. By evaluating both models simmultanously, he interprets that the opening was moved by removing geometry from the model and then adding new geometry to the model that represents the same opening at a different location.



Binder restoration In this example the decision is taken to update the binder by using the new geometry as the opening. Inside the binder instance, the old (removed) geometry is replaced with the newly found geometry. Once the binder is modified the outline of the slab component changes back to a solid thin line, which indicates that the binder is able to perform an update.



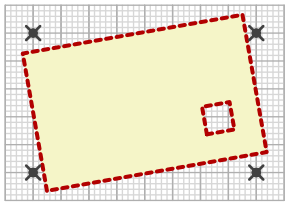
Second slab update The engineer invokes the second update of the example and the slab is updated to match the binding geometry.



Modifying bound components

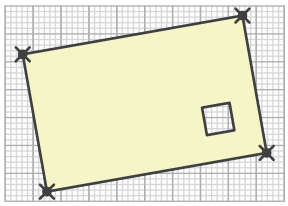
Up to this point in time, the engineer did not modify the slab in a way that violates the defined binding between the slab and the geometry. A pre-requisite of the proposed solution was that it should support the engineer in maintaining consistency, without restricting his flexibility to modify bound objects. If this is not possible, the engineer does not have complete control over the SEM instance that he created.

Changing a bound object The slab is rotated 10° by the engineer. This action violates the binding between the slab and the binding geometry. The outline of the slab changes to a thick dotted line to indicate that the bound object is inconsistent with the binder.



Three possibilities now exist. The first possibility is to perform an update without changing the definition of the binder between the geometry and the slab. The resulting action is that the slab is rotated back to the position that corresponds with the geometry.

The second possibility is to remove the binder instance between the slab and the binding geometry and then to perform an update. The result of this is that the supports are updated to fit to the corner points of the slab again. All subsequent changes to the binding geometry will not be forwarded to the slab instance.



The third option is to adjust the updater of the binder instance in such a way that the update process respects the change that occurred to the slab object and does not change it back to the previous state during an update. Thus, the slab is still bound to the geometry and the update process does

not modify the current state of the slab. After the update the slab remains the same, the supports are moved as in the second scenario and subsequent changes to the geometry are forwarded to the slab instance.

7.7 Pilot application

A pilot application was developed to demonstrate the viability of the solution approach. A brief description of its functionality is given below. This provides the reader with an overview of the effort required to develop a SEM application.

7.7.1 Pilot application: structures@CADEMIA

The CADEMIA extension, structures@CADEMIA, is an add-on application that was developed to create and operate on SEM instances. This package integrates seamlessly with CADEMIA and does not require any modifications to the CADEMIA kernel. It contains the following elements:

- SEM components that are implemented in a standalone application which is independent of the CADEMIA project.
- CADEMIA proxy components that wrap SEM components. These components are transient and created when a SEM model is loaded into CADEMIA in order to provide CADEMIA with access to the SEM components.
- Custom commands that allow the creation of SEM components and binder instances.
- Functionality to derive a FEM instance from a SEM instance.
- A Finite Element application which is independent from CADEMIA and is used to perform the finite element analyses of the SEM instance.
- Finite element meshing functionality.
- Useful utility classes.

The scope of the pilot implementation is to create a SEM instance of an arbitrary shaped floor slab based on some existing geometry and to bind the SEM to this geometry. The SEM is then used to derive a FEM instance of the slab under consideration.

A detailed description of the use of CADEMIA falls outside the scope of this dissertation. The functionality available in a standard CADEMIA runtime instance is also available in the structures@CADEMIA add-on package. For example copying, moving, scaling, mirroring and deleting functionality remains the same in both application instances.

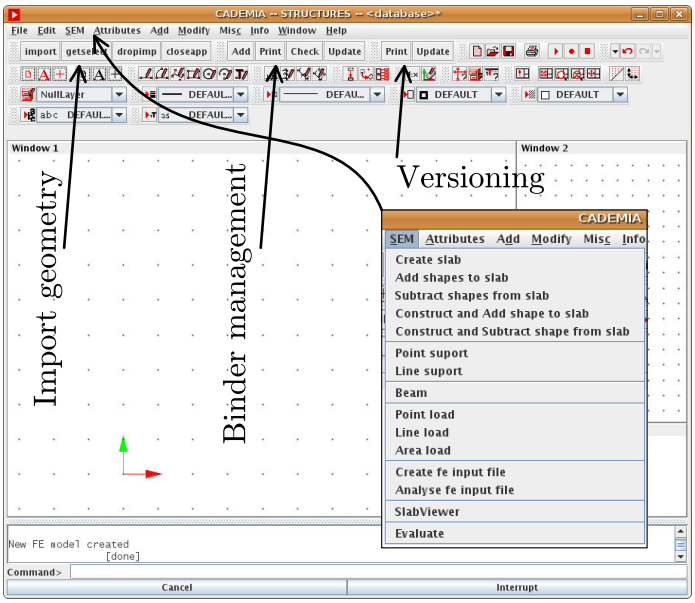


Fig. 7.6: structures@CADEMIA user interface

7.7.2 User interface

Figure 7.6 shows the user interface of structures@CADEMIA. The basic layout is the same as that of a standard CADEMIA instance. One menu and three toolbars are added to the user interface. The newly added menu, i.e. *SEM*, contains the functionality to create the SEM components that are part of the wrapped SEM model.

SEM menu

A short description of each menu command is given to clarify the functionality of the pilot application.

Create slab command creates a slab instance. Before this command is executed, the geometry that forms the boundary of the slab must be selected. The result of the command is a SEM slab component wrapped inside a CADEMIA Slab proxy object. The proxy component allows CADEMIA to display and access the SEM slab component.

Add shapes to slab command adds selected shapes to the slab instance. The selected shapes need to be closed shapes. Some shapes, e.g. bezier curves are closed automatically by connecting their start and end points.

Subtract shapes from slab command subtracts selected shapes from the slab instance. The selected shapes also need to be closed shapes.

Construct and Add shape to slab command joins the selected geometry into one shape and then adds the resultant shape to the slab instance.

Construct and Subtract shape from slab command also joins the selected geometry into one shape and then subtracts the resultant shape from the slab instance.

Point support creates a point support. If control points are preselected, a support is created at each preselected control point and bound to the point, otherwise one point support is created at a location that the user specifies after the command is issued.

Line support creates a continuous line support along a pre-selected shape.

Beam creates a downstand beam that is based on preselected geometry. For the purposes of the pilot application the cross-section of the beam is not modifiable.

Point load creates a point load. If control points are preselected, a point load is created at each pre-selected control point, otherwise one point load is created at a location specified after the command is issued.

Line load creates a line load based on pre-selected geometry.

Area Load creates an area load(s) based on pre-selected geometry. The pre-selected geometry must be one or more closed shapes.

Create Fe input file command creates the text-based input file that is used by the finite element application that is part of the CADEMIA add-on package.

Analyse fe input file processes an existing fe-input file and performs a finite element analysis.

Slab viewer starts the slab viewer to display the results of a finite element analysis.

Evaluate combines the creation of the FEM instance, the analysis and starting the slab viewer. It first prompts for a name of the analysis and for a meshing index. The meshing index is used to control the granularity of the mesh. A meshing index of 500 implies that the area of an undeformed finite element triangle is 500 times smaller than the area of the slab to be meshed. The actual number of elements in the mesh may vary significantly from the index value due to the stretching and refinement of the mesh to match the actual geometry of the slab.

Import toolbar

This toolbar manages the import of geometry. The *import* button starts a separate CADEMIA instance which acts as a source of geometry for the

SEM instance, see figure 7.7. The *getselect* button retrieves the selected geometry in the second CADEMIA instance and imports it into the SEM application. This geometry is now used to define the SEM component instances. The *dropimp* button removes the imported geometry from the SEM application. Once the SEM components are created, the imported geometry is not required in the SEM application anymore and can be discarded. The *closeApp* button closes the second CADEMIA instance that is used for the definition of geometry which the SEM application imports.

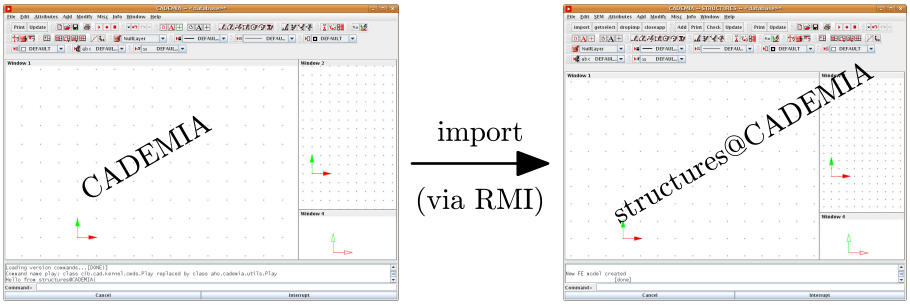
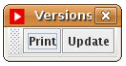


Fig. 7.7: Geometry import source

Binder toolbar

This toolbar manages the binders that bind components of the SEM instance to components of the imported geometry. The *add* button is used to add a user-defined dependency. The user defined binder is created by preselecting the binding objects, executing the command and then selecting the bound object. The default `Updater` associated with the `Binder` created by this command prints a message on the console when the update method is invoked. This is useful for testing purposes. The *print* command displays all the binder instances in the console. The *check* command diplays the update order of the components to be updated and the *update* command invokes the update process.

Versioning toolbar



This toolbar is used inside the CADEMIA instance which provides the versioned geometry to the structures@CADEMIA instance in order to support change detection. The *print* command prints a table that displays the names of the objects, the version numbers and whether the object's state was changed since the version number was assigned. The *update* command is executed when the user

is satified with the changes that were accumulated since the last version event and wants to formally version the new states of the changed objects. The version numbers of all the objects that were changed since the previous versioning event are incremented by one.

7.7.3 Step-by-step example

An example which demonstrates the solution approach, specifically how the binding mechanism functions in a working application, is presented in Appendix A.1.

Chapter 8

Implementation Issues

Important aspects of the architecture of the SEM pilot application are discussed in this chapter. This is done to clarify the solution techniques, especially the binding and updating mechanism, at the level of application development. Java is used as the implementation language.

SEM models have to be linked to CAD models as described in section 4.9. The CAD proxy design pattern described there enables the utilization of an existing CAD system as GUI for domain specific models. The SEM application, in turn, spawns input for the FEM application. Each of the three application types, namely CAD, SEM and FEM, are involved in the pilot application. The open source CAD platform CADEMIA is used in the role of the CAD system. A FEM application was developed by the author, as was the SEM application which represents the middleware model between CADEMIA and the FEM application.

Figure 8.1 presents the parts that form the pilot application. Each part is encapsulated in a Java archive (.jar-file). The CADEMIA and the SEM-application parts are independent of one another. However, *structures@CADEMIA* depends on both CADEMIA and the SEM-application, which means that a running instance of *structures@CADEMIA* requires cademia.jar and sem-app.jar in the classpath of its Java Virtual Machine.

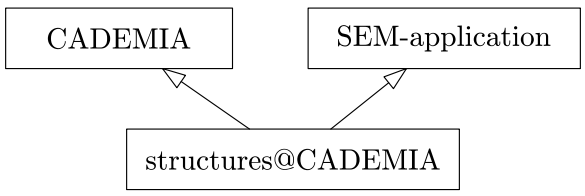


Fig. 8.1: Pilot application parts

structures@CADEMIA contains the CAD commands and components that enable CADEMIA to operate on SEM model instances.

8.1 Using CADEMIA as CAD system

An in-depth description of the internal working of the CADEMIA platform falls outside the scope of this dissertation. This section describes only the core concepts required to extend CADEMIA for the purposes of the pilot application.

Figure 8.2 [Firmenich 2006] shows the most important classes comprising the CADEMIA system. The `Kernel` class is the core of this application.

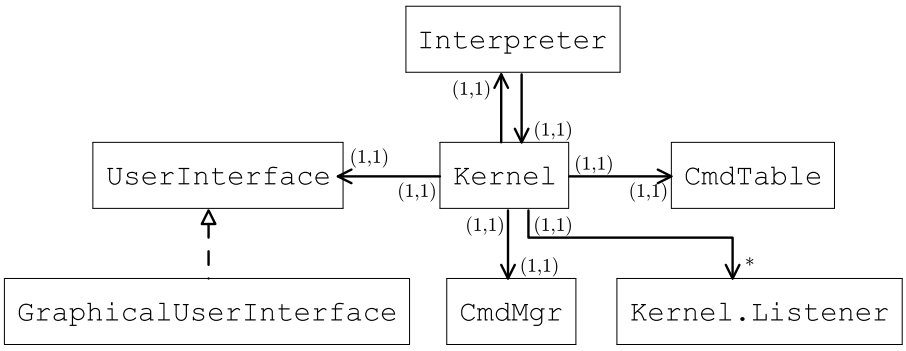


Fig. 8.2: CADEMIA parts

8.1.1 CADEMIA Kernel

The `Kernel` provides a method, `addCmd(String cmdTxt, Class cmdClass)`, by which a command is added to the command table. The command table maps command names to command classes. A command class may have several names mapped to it, e.g. the command class that shuts CADEMIA down is the target of both the commands *quit* and *exit*. Thus, when a user enters either *quit* or *exit* into the command line interpreter, the shutdown command is executed and CADEMIA shuts down.

The command manager, i.e `CmdMgr`, maintains a list of the executed commands and manages the undo and redo functionality of CADEMIA. The interpreter tokenizes the input stream into commands and arguments and passes the commands to the Kernel to be executed.

8.1.2 CADEMIA commands

The command interface `cib.util.cmd.Cmd` specifies the functionality of a `Command` object.

All functionality in CADEMIA is command-based. Commands are responsible for:

- loading and storing of drawing models.

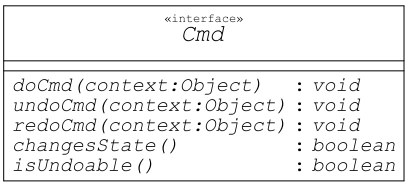


Fig. 8.3: Command interface

- creation of drawing components, e.g. lines.
- manipulation of drawing components, e.g. the rotation of lines.
- removal of drawing components, e.g. the removal of lines.
- manipulation of view, e.g. zooming and panning.
- customization of menus, toolbars and mouse strokes.
- closing the application.

Undoable commands have the ability to capture the state of components that are modified during their execution in order to reverse the changes if the user performs an *undo* on the system. For example, the `Remove` command remembers the components it removed and if the *undo* command is invoked, the components are added back to restore the database to the state it had before the `Remove` command was issued.

8.1.3 CADEMIA components

CADEMIA components all implement the `Component` interface, either directly or by extending the `ComponentAdapter` class.

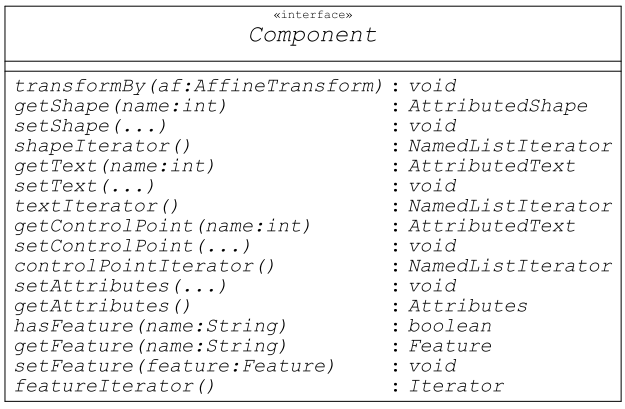


Fig. 8.4: Component interface

The `transformBy(AffineTransform af)` method of this interface allows the geometry of components to be manipulated.

The `shapeIterator()` method gives the panel that displays the components graphically, access to the shape(s) of the components. This method is invoked by the drawing system of CADEMIA to render the drawing components. Likewise, the `textIterator()` is invoked in order to display any text associated with a component. *Features* are used to store additional information of a component, for example the angle of a line or the fill pattern of a rectangle.

Components have control points which allow the manipulation of geometry on the basis of selected control points, instead of the complete component. For example, the two end points of a line component are its control points and if the line is selected, the two control points become visible. If the user only wants to move one end point, that particular control point is selected and moved by executing the `Move` command.

8.1.4 Coordinate spaces

Two two-dimensional Cartesian coordinate spaces exist in CADEMIA. These are the world coordinate space, W, and the user coordinate space, U. The world coordinate space is used to represent the geometry of all components internally. Each point in this space is represented with equal accuracy by normalizing values to the ranges -1 to +1 along both axes. Coordinates in the user coordinate system are transformed to the world coordinate system by a transformation which depends on the overall dimensions of the CAD model.

8.2 SEM-application

This part of the pilot application contains the SEM components that were implemented, as well as functionality to create a finite element mesh and invoke an analysis of a SEM model instance. Some viewing functionality is also included to enable visualization of SEM model instances independently from CADEMIA.

8.2.1 SEM component

A SEM component consists of two parts, the first part is the geometry of the component and the second part is the structural semantics of the component. Figure 8.5 presents the class hierarchy of the implemented SEM components.

Engineering semantics

All the SEM components extend the abstract class `SemComponent`. This class provides its subclasses with functionality to store and retrieve properties in the form of key-value pairs.

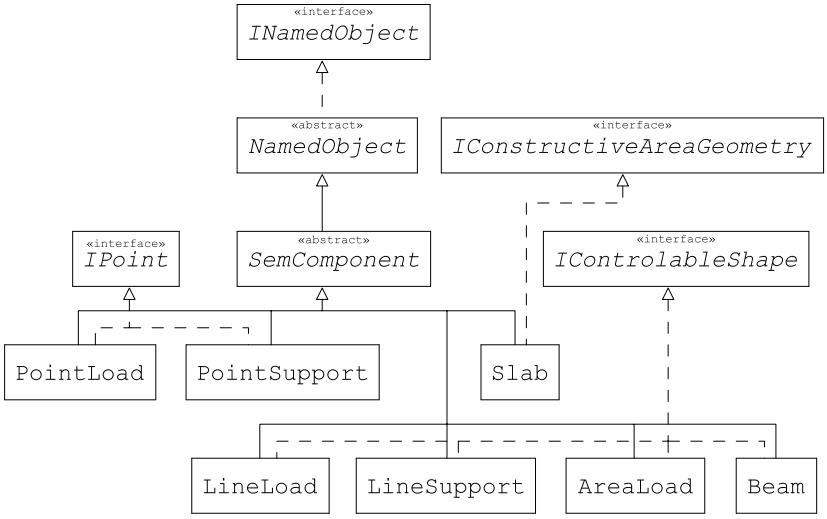


Fig. 8.5: SEM Components

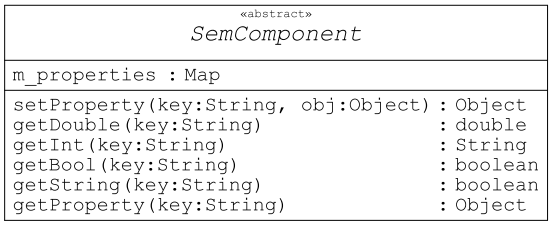


Fig. 8.6: Class SemComponent

Class `SemComponent` extends `NamedObject`, which implies that all instances of `SemComponent`s are persistently identified. The pilot application does not allow manipulation of the non-geometric properties of the SEM components because that has no bearing on the focus of the dissertation.

Geometry

The geometry of SEM components is encapsulated by the specific implementation of the individual SEM components. In the case of SEM components which do not have physical dimensions, e.g. `PointLoad`s and `PointSupport`s, the geometric property of importance, namely their location, is contained inside a `Point2D` instance wrapped by the component. These two components implement the interface `IPoint` which provides the `getPoint()` and `setPoint(Point2D p2d)` methods. This reduces the complexity of the `Updater` implementation drastically, as shown later in this chapter.

SEM components of dimension 1 or higher are divided into two categories. The first category contains components that are described by 1D

lines of arbitrary shape. All the implemented SEM components that have dimension, except the `Slab` component, fall into this category. The `Slab` component falls in the second category, i.e. components described by constructive area geometry (CAG). The `Slab` is the only component in this category.

Three SEM components are discussed in more detail below, namely `PointSupport`, `LineLoad` and `Slab`.

PointSupport

The `PointSupport` class wraps a `Point2D` instance which contains the location of the support. The only methods implemented by this class are the two methods specified in the `IPoint` interface, namely `getPoint()` and `setPoint(Point2D pnt)`, as well as the `Externalizable` interface's methods required to read/write the geometric properties from/into a byte stream. All other functionality is inherited from the super class `SemComponent`.

The `setPoint(Point2D pnt)` fires a notification each time the coordinates of the wrapped `Point2D` change. This change event is propagated to all registered listeners.[1] Change notification is primarily used to update the views of the CAD system. It is also used by `ObserverBased` binders to detect changes to binding objects.

```
package aho.sem.comp;

import ...

public class PointSupport extends SemComponent implements IPoint{

  private static final long serialVersionUID = 1L;

  private Point2D m_pnt = new Point2D.Double();

  public PointSupport(){
    super();
  }

  public PointSupport(String name, Point2D pnt) {
    super(name);
    this.m_pnt.setLocation(pnt);
  }

  public Point2D getPoint(){
    return m_pnt;
  }

  public void setPoint(Point2D pnt){
    this.m_pnt.setLocation(pnt);
    _notifyWasChanged();
  }

    //..... Externalizable .....................
  public void readExternal(ObjectInput in) throws IOException,
      ClassNotFoundException {
    super.readExternal(in);
    m_pnt.setLocation(in.readDouble(), in.readDouble());
  }
  public void writeExternal(ObjectOutput out) throws IOException {
    super.writeExternal(out);
    out.writeDouble(m_pnt.getX());
    out.writeDouble(m_pnt.getY());
  }
}
```

[1] See section 6.3.1 for an explanation of the listener design pattern.

Listing 8.1: SEM PointSupport

LineLoad

The `LineLoad` follows the same implementation pattern as the `PointSupport` described above. The only difference is that the geometry of the component is managed by a `IControlableShape` and not by a `Point2D`.

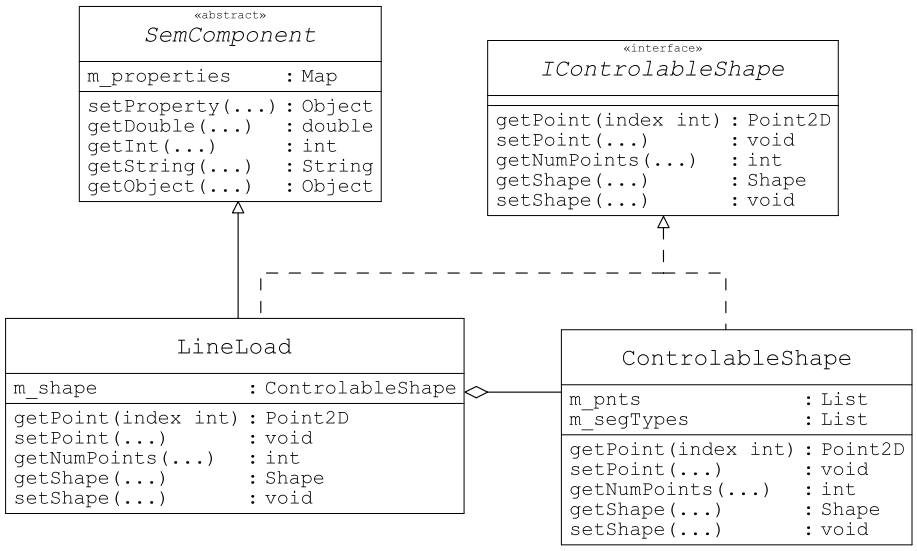


Fig. 8.7: Geometry contol by delegation

Figure 8.7 presents the design pattern that is used to delegate the geometric functionality of the `LineLoad`. The `LineLoad` wraps an instance of an `IControlable` shape while also implementing the `IControlable` interface. The `LineLoad` implementation of the interface delegates each method to the corresponding method of the wrapped `ControlableShape` instance.

The advantage of the *delegate* pattern is that source code in `ControlableShape` can be used by several classes without any duplication.

```
package aho.sem.comp;

import ...

public class LineLoad extends SemComponent implements IControlableShape{

  private static final long serialVersionUID = 1L;

  private ControlableShape m_shape;

  public LineLoad(){
    super();
  }

  public LineLoad(String name, Shape shape) {
```

```
    super(name);
    m_shape = new ControlableShape(shape);
  }

  public Point2D getPoint(int index){
    return m_shape.getPoint(index);
  }

  public void setPoint(int index, Point2D point){
    m_shape.setPoint(index, point);
  }

  public int getNumPoints(){
    return m_shape.getNumPoints();
  }

  public Shape getShape(){
    return m_shape.getShape();
  }

  public void setShape(Shape s){
    m_shape.setShape(s);
    _notifyWasChanged();
  }

    //..... Externalizable .....................
  public void readExternal(ObjectInput in) throws IOException,
      ClassNotFoundException {
    super.readExternal(in);
    m_shape = (ControlableShape)in.readObject();
  }
  public void writeExternal(ObjectOutput out) throws IOException {
    super.writeExternal(out);
    out.writeObject(m_shape);
  }
}
```

Listing 8.2: SEM LineLoad

IControlableShape

The `IControlableShape` interface provides the basic geometric functionality of most of the SEM components (see figure 8.5). The implementation of this interface allows the construction of `Shape` instances of which the points that define the shape's geometry can be accessed / altered on an individual basis.

Listing 8.3 shows a part of the implementation of the `IControlable` interface in class `aho.util.geom.ControlableShape`. Method `setShape( Shape s)` iterates over the shape it receives and populates two lists. The first list, `m_pnts`, contains the points that define the `Shape`. The second list, `m_segTypes` contains the type of segments from which the `Shape` is constructed.

```
package aho.util.geom;

import ...

public class ControlableShape implements IControlableShape, Externalizable{
  private static final long serialVersionUID = 1L;

  private List<Point2D> m_pnts = new ArrayList<Point2D>();
  private List<Integer> m_segTypes = new ArrayList<Integer>();

  public ControlableShape(){
    super();
  }

  public ControlableShape(Shape shape){
    setShape(shape);
  }
```

```
  public Shape getShape(){
    Path2D path2d = new Path2D.Double();
    int _index = 0;
    for(int i = 0; i < m_segTypes.size(); i++){
      switch (m_segTypes.get(i)) {
        case PathIterator.SEG_MOVETO :
          path2d.moveTo(m_pnts.get(_index).getX(), m_pnts.get(_index).getY());
          break;
        case PathIterator.SEG_LINETO:
          path2d.lineTo(m_pnts.get(_index).getX(), m_pnts.get(_index).getY());
          break;
        case PathIterator.SEG_QUADTO:
          path2d.quadTo(m_pnts.get(_index).getX(), m_pnts.get(_index).getY(),
              m_pnts.get(_index+1).getX(), m_pnts.get(_index+1).getY());
          _index = _index + 1;
          break;
        case PathIterator.SEG_CUBICTO :
          path2d.curveTo(m_pnts.get(_index).getX(), m_pnts.get(_index).getY(),
              m_pnts.get(_index+1).getX(), m_pnts.get(_index+1).getY(),
              m_pnts.get(_index+2).getX(), m_pnts.get(_index+2).getY());
          _index = _index + 2;
          break;
        case PathIterator.SEG_CLOSE :
          path2d.closePath();
      }
      _index++;
    }
    return path2d;
  }

  public void setShape(Shape shape){
    PathIterator pi = shape.getPathIterator(null);
    m_pnts.clear();
    m_segTypes.clear();
    double[] tmp = new double[6];
    int type;
    while(!pi.isDone()){
      type = pi.currentSegment(tmp);
      m_segTypes.add(type);
      switch (type) {
        case PathIterator.SEG_MOVETO :
          m_pnts.add(new Point2D.Double(tmp[0],tmp[1]));
          break;
        case PathIterator.SEG_LINETO:
          m_pnts.add(new Point2D.Double(tmp[0],tmp[1]));
          break;
        case PathIterator.SEG_QUADTO:
          m_pnts.add(new Point2D.Double(tmp[0],tmp[1]));
          m_pnts.add(new Point2D.Double(tmp[2],tmp[3]));
          break;
        case PathIterator.SEG_CUBICTO :
          m_pnts.add(new Point2D.Double(tmp[0],tmp[1]));
          m_pnts.add(new Point2D.Double(tmp[2],tmp[3]));
          m_pnts.add(new Point2D.Double(tmp[4],tmp[5]));
          break;
        case PathIterator.SEG_CLOSE :
      }
      pi.next();
    }
  }
  ...
}
```

Listing 8.3: ControlableShape

When the `getShape()` method is invoked, a new `Shape` is constructed using the points and segment types contained in the two lists.

The other methods specified by the `IControlableShape` interface provide access to the individual points contained in the `m_pnts` list.

Slab

The `Slab` component implementation follows the same design pattern as the `LineLoad` described above, the only difference being that the `Slab` class

implements the `IConstructiveAreaGeometry` interface by wrapping a delegate which implements the interface.

```
package aho.sem.comp;

import ...

public class Slab extends SemComponent implements IConstructiveAreaGeometry {

  private static final long serialVersionUID = 1L;

  private IConstructiveAreaGeometry m_area;

  public Slab(){
    super();
  }

  public Slab(String name, Shape s){
    super(name);
    m_area = new AreaConstructor(s);
  }

  public Area getArea(){
    return m_area.getArea();
  }

  public int add(Shape s){
    int index = m_area.add(s);
    _notifyWasChanged();
    return index;
  }

  public int subtract(Shape s){
    int index = m_area.subtract(s);
    _notifyWasChanged();
    return index;
  }

  public void dropPart(int index){
    m_area.dropPart(index);
    _notifyWasChanged();
  }

  public void clear(){
    m_area.clear();
  }

  public void transform(AffineTransform af){
    m_area.transform(af);
    _notifyWasChanged();
  }

  public int getNumCntPnts(){
    return m_area.getNumCntPnts();
  }

  public Point2D getPoint(int index){
    return m_area.getPoint(index);
  }

  public Point2D getPointNo(int index){
    return m_area.getPointNo(index);
  }

  public void setPoint(Point2D p2d, int index){
    m_area.setPoint(p2d, index);
  }

  public void setPointNo(Point2D p2d, int index){
    m_area.setPointNo(p2d, index);
  }

    //..... Externalizable .....
    public void readExternal(ObjectInput in) throws IOException,
      ClassNotFoundException{
      super.readExternal(in);
      m_area = (AreaConstructor)in.readObject();
    }
    public void writeExternal(ObjectOutput out) throws IOException{
      super.writeExternal(out);
      out.writeObject(m_area);
    }
```

```
80 }
```

Listing 8.4: SEM Slab

The wrapped `AreaConstructor` allows the definition of a `Slab` instance by adding and subtracting shapes.

AreaConstructor

`AreaConstructor` implements the `IConstructiveAreaGeometry` interface, and is used by the `Slab` class as the delegate which executes the methods of the interface. The Java class `java.awt.geom.Area` [Java 2007] provides the Constructive Area Geometry (CAG) functionality that is used by the `AreaConstructor`. Thus, before the `AreaConstructor` implementation can be discussed in more detail, class `java.awt.geom.Area` requires a closer look.

An `Area` object stores and manipulates a resolution-independent description of an enclosed area in 2-dimensional space. `Area` objects can be transformed and can perform various CAG operations when combined with other `Area` object instances. The CAG operations include the *addition*, *subtraction*, *intersection*, and *exclusive or* of areas.

Internally, the geometry of the path describing the outline of an `Area` resembles the path from which the area was constructed only in that it describes the same enclosed 2-dimensional area. However, entirely different types and ordering of path segments may be used. An `Area` may use more path segments to describe its outline, even when the original outline is simple and obvious. The analysis that the `Area` class performs on the path is not based on the same concepts of "simple and obvious" as that of a human being.

Figure 8.8 shows an ellipse with a square hole modelled in two different ways. The top part of the figure shows this geometry modelled by an `Area` object instance and the bottom part shows the geometry modelled using the outline shape only. The control points of these shapes are numbered. Both shapes are rotated through 30°. Note that the number of control points of the `Area` based geometry increases and are renumbered e.g. point 13 becomes point 17. The control points of the geometry based on the representation of the outline only are not affected by the transformation.

In order to utilize control points for binding definitions, i.e. allowing a component to be bound to a specific control point of another component, the numbering of control points should not be affected by transformations performed on the components.

Class `AreaConstructor` provides some CAG functionality while keeping the control point numbers consistent. If a shape is added to or subtracted from an `AreaConstructor` instance, the `AreaConstructor` object iterates over the shape and does the following:

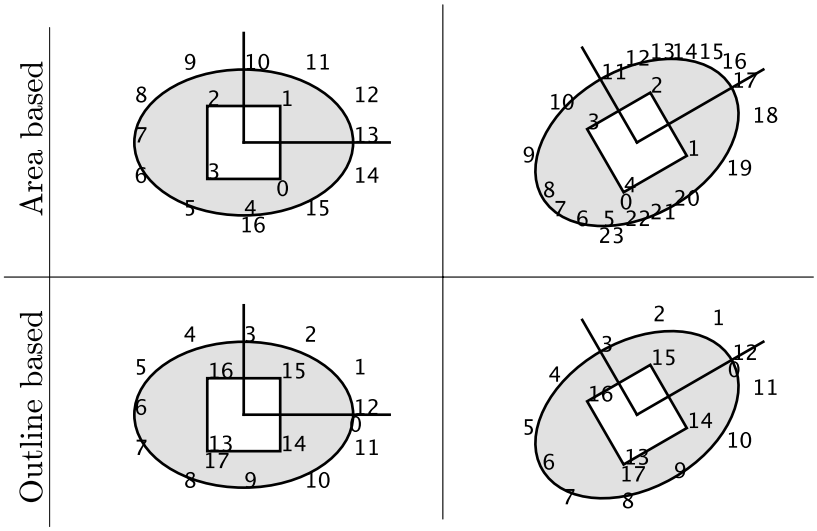


Fig. 8.8: Java awt.geom.Area control points

- retrieve the points that define the `Shape`, by using a `PathIterator`, and assign a unique number to each point retrieved from the shape.
- add the points to a map, using each point's unique number as key.
- represent the shape using two integer arrays. The first array contains the segment types defined in accordance with the `java.awt.geom.PathIterator` interface, i.e. SEG_MOVETO = 0, SEG_LINETO = 1, SEG_QUADTO = 2, SEG_CUBICTO = 3 and SEG_CLOSE = 4. The second array contains the keys, of type integer, of the points that were retrieved from the `Shape`.
- the two arrays are wrapped inside an instance of a `Part` object, which is added to a list that contains all the parts that define the area of the `AreaConstructor`.
- the *type* of CAG operation, i.e. addition or subtraction, is added to a list that stores the operations to be performed in order to construct the resulting area of the `AreaConstructor`.

See figure 8.9 for an example of an AreaConstructor instance.

The storing of the shapes and the operations that combine these shapes in order to define the area of the `AreaConstructor`, rather than the result of the construction process, i.e. an `Area` instance, ensures that the control point numbers remain consistent even if the `AreaConstructor` is transformed. All geometric transformations, e.g. rotate and translate, are performed on the points contained inside the point-map. Whenever the *Shape* of the `AreaConstructor` is required, it is calculated by reassembling

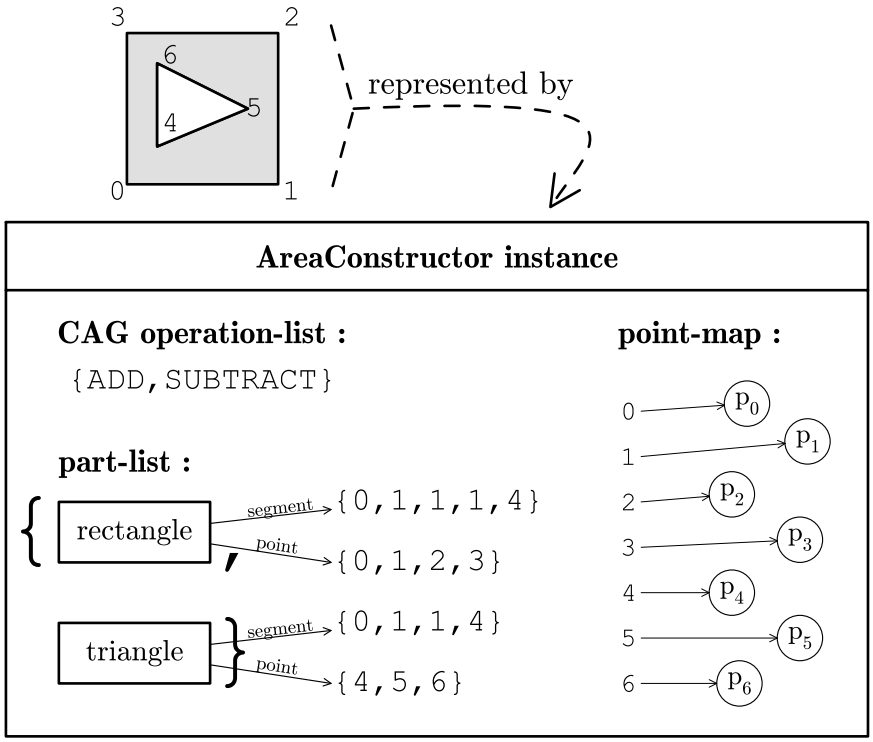


Fig. 8.9: Runtime instance of an AreaConstructor

the source shapes, i.e. by interpreting the two integer arrays contained inside the `Part` instances and performing the stored CAG operations on these shapes. Control points are directly accessible on an individual basis through the mapping of the control point numbers to the control points themselves.

8.3 structures@CADEMIA

The application *structures@CADEMIA* is a CADEMIA extension that is used to test the binding mechanism between CAD and SEM instances. This section explains the implementation details and includes some sections of source code.

The SEM components are divided into two categories, namely single-part components and multi-part components. Single-part components originate from exactly one geometric component. For example, a `LineLoad` is based on the geometry of exactly one line. Multi-part components depend on one or more geometric components. The `Slab` component, for example, allows several different geometric components to be combined in order to define its area.

It is also possible to classify SEM components on the basis of dimension.

A `PointSupport`, for example, is a SEM component with zero dimension, a `LineLoad` is a one dimesional component, and an `AreaLoad` is a two dimensional component.

Before some CAD proxy components are discussed, one class must be briefly introduced: `NamedCademiaObject`. An instance of a `NamedCademiaObject` serves as an object wrapper for a CADEMIA component inside the extended CADEMIA-SEM application (See figure 5.3). It has two attributes i.e. the name of cademia component and a reference to the cademia component itself. This class implements the `INamedObject` interface that allows the binding mechanism to access CADEMIA components via the `INamedObject` interface. NamedCademiaObjects are transient and, like the CAD proxy objects, not stored in the SEM model outside of the CADEMIA context.

The remainder of this section describes individual CAD proxy components with the command classes that are responsible for its instantiation.

8.3.1 Single-part, zero-dimensional component example: PointSupport

The `PointSupport` is an example of a SEM component with zero dimension. The CADEMIA proxy component of a `PointSupport` instance is a `CadPntSupport` instance. The command class that is used to create point supports is `CreatePointSupport`.

Component

A `CadPntSupport` component wraps a SEM `PointSupport` and extends the `ComponentAdapter` class provided by the CADEMIA system. All the information relating to the support is stored inside the `PointSupport` instance that is wrapped by the `CadPntSupport` instance. Therefore, it is not possible for a `CadPntSupport` to exist on its own.

```
package aho.cademia.sem.comp;

import aho.sem.comp.PointSupport;
...
public class CadPntSupport extends CadProxy{//CadProxy extends ComponentAdapter

  private PointSupport m_pointSupport; //SEM component

  private double m_size = .1; // size of graphical representation

  public CadPntSupport(PointSupport pointSupport){
    this.m_pointSupport = pointSupport;
  }

  public INamedObject getEngComp() {
    return m_pointSupport;
  }

  public void transformBy(AffineTransform at) {
    Point2D p2d = CoordinateSystem.fem2cad(m_pointSupport.getPoint(), null);
    Point2D pnt = at.transform(p2d,null);
    m_pointSupport.setPoint(CoordinateSystem.cad2fem(pnt, null));
    _notifyWasChanged();
```

```
25    }

      public Point2D getPoint(){
        return m_pointSupport.getPoint();
      }
30
      public void hasChanged(){
        _notifyWasChanged();
      }

35      // Interface Component
      public Object clone() {
        CadPntSupport ps = new CadPntSupport(
          new PointSupport(NameService.genName("ps"),m_pointSupport.getPoint()));
        return ps;
40    }

      private AttributedShape getShape() {
        // Create the attributed shape
        Point2D p2d = CoordinateSystem.fem2cad(m_pointSupport.getPoint(), null);
45      GeneralPath gp = new GeneralPath();
        gp.moveTo((float)(p2d.getX()-m_size),(float)(p2d.getY()-m_size));
        gp.lineTo((float)(p2d.getX()+m_size),(float)(p2d.getY()+m_size));
        gp.moveTo((float)(p2d.getX()+m_size),(float)(p2d.getY()-m_size));
        gp.lineTo((float)(p2d.getX()-m_size),(float)(p2d.getY()+m_size));
50      AttributedShape _at = _attributeShape(new AttributedShape(gp));
        getAttributes().setLinePattern("din_strichlinie");
        getAttributes().setLineWidth("din_100");
        getAttributes().setDrawPaint("din_blau");
        return _at;
55    }

      public NamedListIterator shapeIterator() {
        return NamedListIteratorAdapter.singletonNamedListIterator(getShape());
      }
60
      public Point2D getControlPoint(int name) throws IllegalArgumentException {
        if (name == 0) {
          return CoordinateSystem.fem2cad(m_pointSupport.getPoint(), null);
        }
65      throw new IllegalArgumentException();
      }

      public void setControlPoint(Point2D pnt, int name)
        throws UnsupportedOperationException {
70      if (name == 0) {
          m_pointSupport.setPoint(CoordinateSystem.cad2fem(pnt,null));
          _notifyWasChanged();
          return;
        }
75      throw new IllegalArgumentException();
      }

      public NamedListIterator controlPointIterator() {
        return new NamedListIteratorAdapter() {
80        protected int _size() {
            return 1;
          }
          protected Object _get(int index) {
            if(index == 0){
85            return CoordinateSystem.fem2cad(m_pointSupport.getPoint(), null);
            }
            throw new IllegalArgumentException();
          }
          protected void _set(int index, Object value) {
90          Point2D p2d = (Point2D)value;
            if(index == 0){
              m_pointSupport.setPoint(CoordinateSystem.cad2fem(p2d,null));
              _notifyWasChanged();
              return;
95          }
                throw new IllegalArgumentException();
              }
          };
      }
100 }
```

Listing 8.5: CADEMIA CadPntSupport proxy

public void transformBy(AffineTransform at): This method is invoked by the CADEMIA system if geometry is transformed. The only geometric property of a point support is it's location. Thus, this method retrieves the location of the point support from the wrapped `PointSupport` instance, transforms the location and stores it back into the `PointSupport` instance.

The coordinate system used in the CAD/SEM models may not be the same as that of the FEM model. Consequently the `CoordinateSystem` class provides methods to transform points between the CAD/SEM coordinate system and the FEM coordinate system.

private AttributedShape getShape(): This private method is a helper method to construct the shape that represents a point support graphically inside the CADEMIA system. It returns a cross-like shape with some additional attributes, i.e. the line width, colour and line pattern.

public NamedListIterator shapeIterator(): This method returns a single `AttributedShape` inside a `NamedListIterator`. The shape is constructed by the `getShape()` method described above and is rendered by the CADEMIA rendering system.

public NamedListIterator controlPointIterator(): This method returns the location of the point support as the only control point of this component. The CADEMIA system invokes this method to display the control points of the selected components. The control point information is based on the location of the point support only and always stored inside and retrieved from the the wrapped `PointSupport` instance.

Command

The class responsible for the creation of `CadPntSupport` instances is shown below. This class is used to create a single support instance at a specific location identified by the user after the command was issued. Alternatively it creates a set of supports, one at each preselected control point with a corresponding binder instance between the wrapped SEM `PointSupport` and the preselected control point. The functionality of this class is briefly described below.

```
package aho.cademia.sem.cmds;

import aho.cademia.sem.comp.CadPntSupport;
...

public class CreatePointSupport implements Cmd{

  public static final double EPS = 0.01;

  private Set<Component> m_pntProxy = new HashSet<Component>();
  private Set<PointSupport> m_pntSupport = new HashSet<PointSupport>();
```

```
  private Set<NamedCademiaObject> m_wrappedComp =
    new HashSet<NamedCademiaObject>();
  private Set m_binders = new HashSet();
  private Set m_uniquePoints = new HashSet();

  private boolean uniquePoint(Point2D p2d){
    Iterator it = m_uniquePoints.iterator();
    while(it.hasNext()){
      Point2D pnt = (Point2D)it.next();
      if(pnt.distance(p2d)<EPS){
        m_uniquePoints.add(p2d);
        return false;
      }
    }
    m_uniquePoints.add(p2d);
    return true;
  }

  public void doCmd(Object context) throws CmdAbortedException {
    EngModel sem = SemCad.getSEM();
    Kernel krnl = (Kernel)context;
    Database db = krnl.getDatabase();
    NameSpace ns = db.getNameSpace();
    MarkerMap mm = db.getMarkerMap();
    Set selSet = db.getSelectSet();

    if(selSet.isEmpty()){
      Point2D p2d = krnl.readPoint("Indicate point support location");
          PointSupport _ps = new PointSupport(NamingService.genName("ps"),p2d);
      CadPntSupport _cps = new CadPntSupport(_ps);
        db.getComponentSet().add(_cps);
      m_pntSupport.add(_ps);
      m_pntProxy.add(_cps);
    }
    else{
      Iterator it = selSet.iterator();
      while (it.hasNext()) {
        Object o = it.next();
        if (o instanceof Component) {
          Component comp = (Component)o;
          if (mm.hasMarkedPrimitives(comp,MarkerMap.CONTROL_POINT_CHAIN)){
            NamedCademiaObject _ncadObj = null;
            if(!(comp instanceof CadProxy)){
              _ncadObj = SemCad.wrap(ns.getName(comp),comp);
              m_wrappedComp.add(_ncadObj);
              sem.add(_ncadObj);
            }
            NamedListIterator lit = comp.controlPointIterator();
            while (lit.hasNext()) {
              Point2D pnt = (Point2D)lit.next();
              int index = lit.previousIndex();
              if(uniquePoint(pnt) && mm.primitiveMarked(
                  comp,MarkerMap.CONTROL_POINT_CHAIN, index)){
                PointSupport _ps
                  = new PointSupport(NamingService.genName("ps"),pnt);
                CadPntSupport _cps = new CadPntSupport(_ps);
                db.getComponentSet().add(_cps);
                sem.add(_ps);
                NamedCademiaObject _ncps = SemCad.wrap(ns.getName(_cps),_cps);
                IBinder _s2compBndr = null;
                if(_ncadObj != null){
                  _s2compBndr = new VersionbasedBinder(_ps,_ncadObj,
                    new PointShapeUpdater(index));
                }
                else{
                  _s2compBndr = new VersionbasedBinder(_ps,
                    ((CadProxy)comp).getEngComp(),new PointShapeUpdater(index));
                }
                sem.add(_s2compBndr);
                sem.add(_ncps);
                      //undo / redo information
                m_pntSupport.add(_ps);
                m_pntProxy.add(_cps);
                m_wrappedComp.add(_ncps);
                m_binders.add(_s2compBndr);
              }
            }
          }
        }
      }
    }
  }

  public void redoCmd(Object context) {
```

```
      EngModel sem = SemCad.getSEM();
      Kernel krnl = (Kernel) context;
      Database db = krnl.getDatabase();
      db.getComponentSet().addAll(m_pntProxy);
100   Iterator<PointSupport> _semIter = m_pntSupport.iterator();
      while(_semIter.hasNext()){
        sem.add(_semIter.next());
      }
      Iterator<NamedCademiaObject> _nIter = m_wrappedComp.iterator();
105   while(_nIter.hasNext()){
        sem.add(_nIter.next());
      }
      Iterator<IBinder> _bndrIter = m_binders.iterator();
      while(_bndrIter.hasNext()){
110     sem.add(_bndrIter.next());
      }
    }

    public void undoCmd(Object context) {
115   EngModel sem = SemCad.getSEM();
      Kernel krnl = (Kernel) context;
      Database db = krnl.getDatabase();
      db.getComponentSet().removeAll(m_pntProxy);
      Iterator<PointSupport> _semIter = m_pntSupport.iterator();
120   while(_semIter.hasNext()){
        sem.remove(_semIter.next());
      }
      Iterator<NamedCademiaObject> _nIter = m_wrappedComp.iterator();
      while(_nIter.hasNext()){
125     sem.remove(_nIter.next());
      }
      Iterator<IBinder> _bndrIter = m_binders.iterator();
      while(_bndrIter.hasNext()){
        sem.remove(_bndrIter.next());
130   }
    }

    public boolean changesState() {
      return true;
135 }

    public boolean isUndoable() {
      return true;
    }
140 }
```

Listing 8.6: Create pointsupport command

private Set m_psupport: This set contains the references of the `CadPnt-Support` instances that are created by the command. This information is required by the *undo* and *redo* methods.

private Set m_bindings: This set contains the references of the `Binder` instances that binds the `CadPntSupport` instances to the preselected control points if applicable. Note that one `Binder` instance is created for each control point to `CadPntSupport` mapping. This set is also required by the *undo* and *redo* methods.

private Set m_uniquePoints: This set is used to keep track of the actual locations where `CadPntSupport` instances were created.

private boolean uniquePoint(Point2D p2d): This method is used to detect preselected control points that coincide with one another. Thus, when a user selects two control points that share the same location, only

one `CadPntSupport` is created and bound to the first control point found at the specific location. This avoids the introduction of duplicate components without the user's knowledge.

public void doCmd(Object context) throws CmdAbortedException: This method receives a reference to the `Kernel` instance which establishes access to the surrounding CADEMIA environment. Firstly, the *context* parameter is cast into a `Kernel` reference. This makes the methods provided by the `Kernel` available. The `Database`, i.e. the container of the CAD components, is retrieved via the `Kernel` and the *marker_map* and *select_set* are obtained via the reference to the `Database`. Next, depending on whether control points were preselected or not the following happens:

If the select set is empty, the method requests a point from the `Kernel` at which a single point support is to be created. In this case the `PointSupport` instance is not bound to any geometry thus, no `Binder` instance is created.

If the select set is not empty, all the selected components and their selected control points are processed individually. Firstly, a selected component is wrapped inside a `NamedCademiaObject` instance if and only if it is not of type `CadProxy`. The `NamedCademiaObject` ensures that the selected native CADEMIA component can be handled by the implemented binding mechanism. Next, the uniqueness of the location is checked in order to ensure that no supports are created on top of one another by the execution of a single command. A SEM `PointSupport` instance is created and wrapped by a newly created `CadPntSupport` instance at each unique location with a corresponding `Binder` instance to support consistency.

The creation of the `Binder` instance depends on the component type to which the control point belongs, see figure 8.10. If the control point belongs to a `CadProxy` instance, the `Binder` is created between the wrapped SEM component and the `PointSupport` instance. Otherwise, the `Binder` is created between the `NamedCademiaObject` instance and the `PointSupport`. This enables the binding mechanism to bind SEM components directly to other SEM components or to native CADEMIA components.

The created `CadPntSupport`s and `Binder` instances are added to the sets described above to provide information for the *undo* and *redo* methods.

public void undoCmd(Object context): This method removes the `CadPntSupport` instances that were created by the `doCmd(Object context)` method from the `Database`. It also removes the `Binder` instances from the binding layer. After this method has been invoked, the state of the `Database` is the same as before the `doCmd(Object context)` was executed.

public void redoCmd(Object context): This method adds the originally created `CadPntSupport` instances back into the cad `Database`. It also

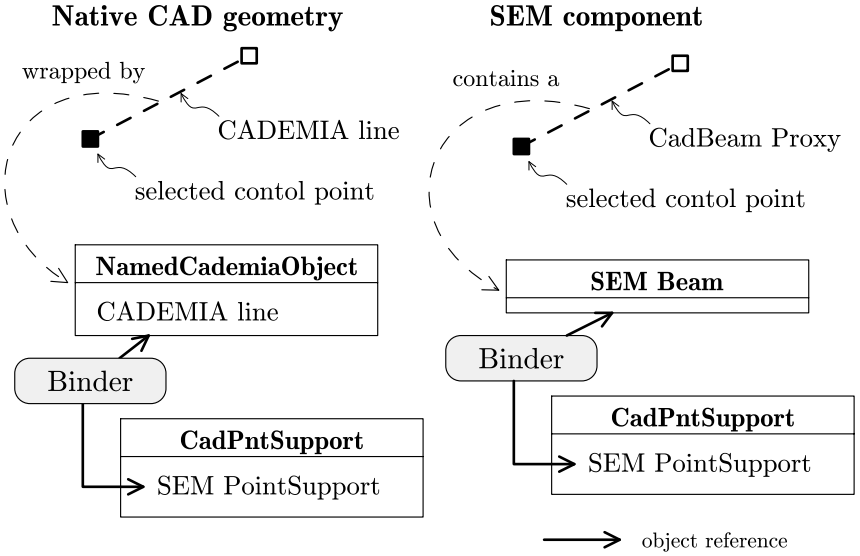


Fig. 8.10: Binder approach: native vs proxy geometry

restores the state of the binding mechanism by adding the previously created `Binder` instances back into the binding layer. After this method is invoked, the state of the `Database` is the same as after the initial execution of the `doCmd(Object context)` method.

8.3.2 Single-part, one-dimensional component example: LineLoad

The SEM component `LineLoad` is represented by the `CadLnLoad` class inside CADEMIA. The command class responsible for the creation of `CadLnLoad` instances is `CreateLineLoad`.

Component

The component implementation is similar to the implementation of the `CadPntSupport` described above. Some functionality is added to support the management of the control points. The source code is presented and described below.

```
package aho.cademia.sem.comp;

import aho.sem.comp.LineLoad;
...
public class CadLnLoad extends CadProxy{
  private static final long serialVersionUID = 1L;
  private LineLoad m_linesupport; // fem coordinate system
  public CadLnLoad(LineLoad lineLoad){
```

```
    this.m_linesupport = lineLoad;
  }

  public INamedObject getEngComp() {
    return m_linesupport;
  }

  private AttributedShape getShape() {
    // Create the attributed shape
    AttributedShape _at = _attributeShape(new AttributedShape(
    CoordinateSystem.fem2cad(m_linesupport.getShape())));
    getAttributes().setLinePattern("din_strichlinie");
    getAttributes().setLineWidth("din_070");
    getAttributes().setDrawPaint("din_rot");
    return _at;
  }

  public void setShape(Shape shape){
    Shape _shape = CoordinateSystem.cad2fem(shape);
    m_linesupport.setShape(_shape);
    _notifyWasChanged();
  }

  public Object clone() {
    CadLnLoad comp = new CadLnLoad(new LineLoad(NamingService.genName("11"),
      m_linesupport.getShape()));
    return comp;
  }

  public NamedListIterator controlPointIterator() {
    return new NamedListIteratorAdapter() {
      protected int _size() {
        return m_linesupport.getNumPoints();
      }
      protected Object _get(int index) {
        return CoordinateSystem.fem2cad(m_linesupport.getPoint(index),null);
      }
      protected void _set(int index, Object value) {
        Point2D pnt = (Point2D) value;
        m_linesupport.setPoint(index, CoordinateSystem.
        cad2fem(new Point2D.Double(pnt.getX(),pnt.getY()),null));
        _notifyWasChanged();
      }
    };
  }

  public Point2D getControlPoint(int name) throws IllegalArgumentException {
    return CoordinateSystem.fem2cad(m_linesupport.getPoint(name),null);
  }

  public void setControlPoint(Point2D pnt, int name)
      throws UnsupportedOperationException {
    m_linesupport.setPoint(name, CoordinateSystem.
      cad2fem(new Point2D.Double(pnt.getX(),pnt.getY()),null));
    _notifyWasChanged();
  }

  public NamedListIterator shapeIterator() {
    return NamedListIteratorAdapter.singletonNamedListIterator(getShape());
  }

  public void transformBy(AffineTransform mat) {
    AffineTransform trf = CoordinateSystem.fem2cadTransform();
    trf.preConcatenate(mat);
    trf.preConcatenate(CoordinateSystem.cad2femTransform());
    Shape _s = trf.createTransformedShape(m_linesupport.getShape());
    m_linesupport.setShape(_s);
    _notifyWasChanged();
  }
}
```

Listing 8.7: CADEMIA CadLnLoad proxy

private LineLoad m_lineLoad : This attribute references the wrapped SEM `LineLoad` component, which encapsulates all the lineload related information. From the perspective of the CAD system, the `CadLnLoad` instance accesses the geometry of the SEM `LineLoad` by invoking the `getShape()`

and `setShape(Shape shape)` methods provided by the SEM `LineLoad`.

public NamedListIterator controlPointIterator(): This method returns a `NamedListIterator` that retrieves the control point information directly from the wrapped `LineLoad` instance.

Command

The source code responsible for the creation of `CadLnLoad` instances is shown below. When this command is invoked, it obtains a reference on the select set of the CAD database. For each preselected shape found in the select set, a SEM `LineLoad` is created and wrapped by a `CadLnLoad` instance. These `CadLnLoad` instances are added to the database of the CAD system.

```
package aho.cademia.sem.cmds;

import aho.sem.comp.LineLoad;
...

public class CreateLineLoad implements Cmd {

  private Set<Component> m_lnProxy = new HashSet<Component>();
  private Set<LineLoad> m_lnLoad = new HashSet<LineLoad>();
  private Set<NamedCademiaObject> m_wrappedComp =
    new HashSet<NamedCademiaObject>();
  private Set m_binders = new HashSet();

  public void doCmd(Object context) throws CmdAbortedException {
    EngModel sem = SemCad.getSEM();
    Kernel krnl = (Kernel)context;
    Database db = krnl.getDatabase();
    NameSpace ns = db.getNameSpace();
    Set selSet = db.getSelectSet();

    Iterator iter = selSet.iterator();
    while(iter.hasNext()){
      Component comp = (Component)iter.next();
      NamedCademiaObject _ncadObj = null;
      if(!(comp instanceof CadProxy)){
        _ncadObj = SemCad.wrap(ns.getName(comp),comp);
        m_wrappedComp.add(_ncadObj);
        sem.add(_ncadObj);
      }
      Iterator _it = comp.shapeIterator();
      while(_it.hasNext()){
        LineLoad _ll = new LineLoad(NamingService.genName("ll"),
          (Shape)_it.next());
        CadLnLoad _cps = new CadLnLoad(_ll);
        db.getComponentSet().add(_cps);
        sem.add(_ll);
        IBinder _ll2compBndr = null;
        if(_ncadObj != null){
          _ll2compBndr = new VersionbasedBinder(_ll,_ncadObj,
            new ShapeUpdater());
        }else{
          _ll2compBndr = new VersionbasedBinder(_ll,
            ((CadProxy)comp).getEngComp(),new ShapeUpdater());
        }
        sem.add(_ll2compBndr);;
        //undo / redo information
        m_lnLoad.add(_ll);
        m_lnProxy.add(_cps);
        m_binders.add(_ll2compBndr);
      }
    }
  }

  public void redoCmd(Object context) {
    EngModel sem = SemCad.getSEM();
    Kernel krnl = (Kernel) context;
    Database db = krnl.getDatabase();
    db.getComponentSet().addAll(m_lnProxy);
```

```
60      Iterator<LineLoad> _semIter = m_lnLoad.iterator();
        while(_semIter.hasNext()){
          sem.add(_semIter.next());
        }
        Iterator<NamedCademiaObject> _nIter = m_wrappedComp.iterator();
65      while(_nIter.hasNext()){
          sem.add(_nIter.next());
        }
        Iterator<IBinder> _bndrIter = m_binders.iterator();
        while(_bndrIter.hasNext()){
70        sem.add(_bndrIter.next());
        }
    }

    public void undoCmd(Object context) {
75    EngModel sem = SemCad.getSEM();
      Kernel krnl = (Kernel) context;
      Database db = krnl.getDatabase();
      db.getComponentSet().removeAll(m_lnProxy);

80    Iterator<LineLoad> _semIter = m_lnLoad.iterator();
      while(_semIter.hasNext()){
        sem.remove(_semIter.next());
      }
      Iterator<NamedCademiaObject> _nIter = m_wrappedComp.iterator();
85    while(_nIter.hasNext()){
        sem.remove(_nIter.next());
      }
      Iterator<IBinder> _bndrIter = m_binders.iterator();
      while(_bndrIter.hasNext()){
90      sem.remove(_bndrIter.next());
      }
    }

    public boolean changesState() {
95    return true;
    }

    public boolean isUndoable() {
      return true;
100 }
}
```

Listing 8.8: Create lineload command

public void doCmd(Object context) throws CmdAbortedException: This method receives a reference to the `Kernel` instance which establishes access to the surrounding CADEMIA environment. Firstly, the *context* parameter is cast into a `Kernel` reference. This makes the methods provided by the `Kernel` available. The `Database` is retrieved via the `Kernel` and the *namespace* and *select_set* are obtained via the reference to the `Database`.

An iterator over the *select_set* returns the selected components one-by-one. Each of the selected components is used to create a `LineLoad` instance with a corresponding `CadLnLoad`, i.e. the CAD proxy component.

The selected components can be divided into two categories, see figure 8.11, namely native CADEMIA components and proxy components. If the selected component is of instance `CadProxy`, a `Binder` instance is created that binds the newly created `LineLoad` to the native SEM component contained inside the `CadProxy` component. If the selected component is a native CADEMIA component, the component is wrapped inside a `NamedCademiaObject`. A Binder instance is then created between the `NamedCademiaObject` and the newly created `LineLoad`.

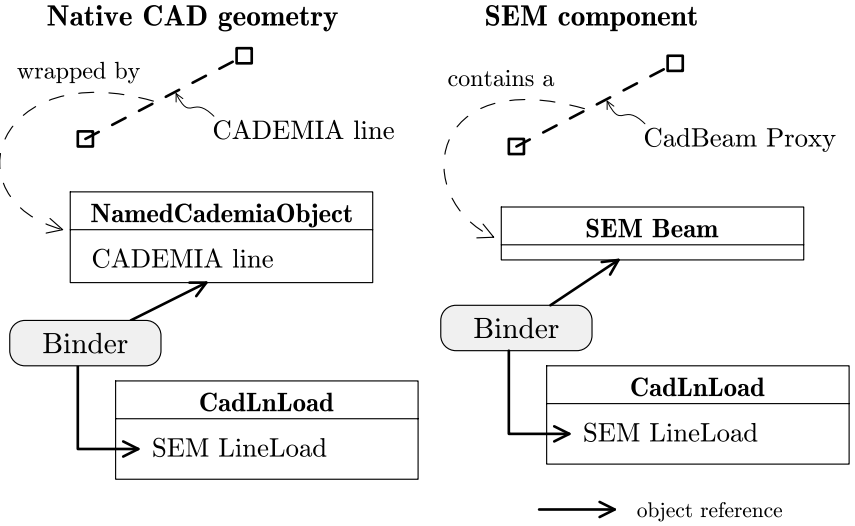


Fig. 8.11: Binder approach: native vs proxy geometry

This command is undoable, therefore it adds the created `CadLnLoad`s, SEM `LineLoad`s, `NamedCademiaObject`s and `Binder` instances to the corresponding private sets: m_lnProxy, m_lnLoad, m_wrappedComp and m_binders to provide information for the *undo* and *redo* methods.

8.3.3 Single-part, two-dimensional component example: AreaLoad

The outline of the geometry of a single-part 2D component is described by a closed shape. This implies that the 2D component has an area greater than zero.

By only describing the outline of a 2D single-part component, the same implementation as the one-dimensional component described above is utilized, with the only additional requirement that the geometry must be closed. Therefore the added classes to CADEMIA are similar. The only difference between the `AreaLoad` proxy in CADEMIA and the `LineLoad` proxy is that another type of SEM component is wrapped, i.e. an `AreaLoad` instead of a `LineLoad`.

8.3.4 Multi-part, two-dimensional component example: CadSlab

In the same way as before the CAD proxy component `CadSlab` wraps an instance of a SEM component, namely `Slab`. The slab proxy allows a `Slab` instance to be constructed in different steps, and it is possible to add and subtract shapes from the slab proxy. This enables the modelling of slabs

with arbitrary shapes, see figure 8.12.

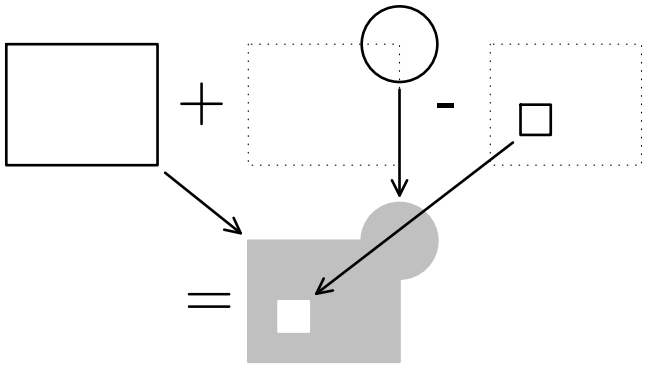


Fig. 8.12: Slab component supports CAG

The pilot implementation allows only one `CadSlab` instance inside the application at a time. This is only done to keep the complexity of the pilot implementation low in order to clearly communicate the way the binding mechanism was implemented.

Component

The implementation of the component is similar to those described above, except that additional methods, `addShape(Shape s)` and `subtractShape(Shape s)` are introduced. These two methods invoke the corresponding methods of the wrapped SEM `Slab` instance and are responsible for the CAG functionality.

```
package aho.cademia.sem.comp;

import aho.sem.comp.Slab;
...

public class CadSlab extends CadProxy{ //CadProxy extends ComponentAdapter

  private static final long serialVersionUID = 1L;

  private Slab m_slab; // fem coordinate system

  public CadSlab(Slab slab){
    this.m_slab = slab;
  }

  public INamedObject getEngComp() {
    return m_slab;
  }

    public Object clone() {
      CadSlab comp = new CadSlab(
        new Slab(NamingService.genName("slab"),m_slab.getArea()));
        return comp;
    }

    public int addShape(Shape shape){
      int index = m_slab.add(shape);
      _notifyWasChanged();
      return index;
    }
```

```
    public int subtractShape(Shape shape){
      int index = m_slab.subtract(shape);
      _notifyWasChanged();
      return index;
    }

    public void dropShape(int index){
      m_slab.dropPart(index);
      _notifyWasChanged();
    }

  private AttributedShape getShape() {
    // Create the attributed shape
    AttributedShape _at = _attributeShape(
      new AttributedShape(CoordinateSystem.fem2cad(m_slab.getArea())));
    getAttributes().setFillPaint("din_beton_bewehrt");
    getAttributes().setLineWidth("din_035");
    return _at;
  }

    public NamedListIterator controlPointIterator() {
        return new NamedListIteratorAdapter() {
            protected int _size() {
                return m_slab.getNumCntPnts();
            }

            protected Object _get(int index) {
                return CoordinateSystem.fem2cad(m_slab.getPointNo(index),null);
            }

            protected void _set(int index, Object value) {
                Point2D p = (Point2D) value;
                m_slab.setPointNo(CoordinateSystem.cad2fem(p, null), index);
                _notifyWasChanged();
            }
        };
    }

    public Point2D getControlPoint(int name) throws IllegalArgumentException {
      return CoordinateSystem.fem2cad(m_slab.getPointNo(name),null);
    }

    public void setControlPoint(Point2D pnt, int name)
        throws UnsupportedOperationException {
      m_slab.setPointNo(CoordinateSystem.cad2fem(
      new Point2D.Double(pnt.getX(),pnt.getY()),null), name);
      _notifyWasChanged();
    }

    public NamedListIterator shapeIterator() {
        return NamedListIteratorAdapter.singletonNamedListIterator(getShape());
    }

    public void transformBy(AffineTransform mat) {
        AffineTransform trf = CoordinateSystem.fem2cadTransform();
        trf.preConcatenate(mat);
        trf.preConcatenate(CoordinateSystem.cad2femTransform());
        m_slab.transform(trf);
        _notifyWasChanged();
    }
}
```

Listing 8.9: CADEMIA CadSlab proxy

Commands

The source code of the command class responsible for the creation of a `CadSlab` instance is shown below. This command is implemented in such a way that only native CADEMIA geometry is used to define the boundary of the `Slab` instance and that exactly one CADEMIA component must be preselected before this command is executed. The motivation of this is again to keep the complexity low so as not to obscure how the binding mechanism is implemented. With some modification this command can be adapted to

combine several pre-selected shapes to form a more complex initial shape for the definition of the `Slab` instance.

```
package aho.cademia.sem.cmds;

import aho.sem.comp.CadSlab;
...

public class CreateSlab implements Cmd {

  private Slab m_slab;
  private CadSlab m_slabProxy;
  private IBinder m_binder;
  private NamedCademiaObject m_compWrapper;

  public void doCmd(Object context) throws CmdAbortedException {
    EngModel sem = SemCad.getSEM();
    Kernel krnl = (Kernel) context;
    Database db = krnl.getDatabase();
    Set selSet = db.getSelectSet();
    NameSpace ns = db.getNameSpace();
    Shape s = null;
    Component comp = null;
    Iterator iter = Collections.filterableIterator(selSet.iterator(),
      new IteratorFilter(){
        public boolean matches(Object o) {
          if(o instanceof CadPntLoad)
            return false;
          if(o instanceof Component)
            return true;
          return false;
        }
    });
    if(iter.hasNext()){
      comp = (Component)iter.next();
      s = (Shape)comp.shapeIterator().next();
    }
    if(iter.hasNext()){
      System.out.println("More than one component selected:" +
        " cannot create slab!");
      return;
    }
    m_slab = new Slab(NamingService.genName("slab"),s);
    m_slabProxy = new CadSlab(m_slab);
    db.getComponentSet().add(m_slabProxy);

    SlabUpdater _updater = new SlabUpdater();
    _updater.add(ns.getName(comp),AreaConstructor.ADD);
    m_compWrapper = SemCad.wrap(ns.getName(comp),comp);
    m_binder = new VersionbasedBinder(m_slab,m_compWrapper,_updater);
    sem.add(m_compWrapper);
    sem.add(m_slab);
    sem.add(m_binder);
  }

  public void redoCmd(Object context) {
    if(m_slabProxy==null)
      return;
    Kernel krnl = (Kernel) context;
    Database db = krnl.getDatabase();
    db.getComponentSet().add(m_slabProxy);
    EngModel sem = SemCad.getSEM();
    sem.add(m_compWrapper);
    sem.add(m_slab);
    sem.add(m_binder);
  }

  public void undoCmd(Object context) {
    if(m_slabProxy==null)
      return;
    Kernel krnl = (Kernel) context;
    Database db = krnl.getDatabase();
    db.getComponentSet().remove(m_slabProxy);
    EngModel sem = SemCad.getSEM();
    sem.remove(m_compWrapper);
    sem.remove(m_slab);
    sem.remove(m_binder);
  }

  public boolean changesState() {
    return true;
  }
```

```
80
    public boolean isUndoable() {
      return true;
    }
}
```

Listing 8.10: Create slab command

public void doCmd(Object context) throws CmdAbortedException: This method firstly checks that one CADEMIA component is selected and that this component is not a CadProxyComponent. This is done with the aid of a custom iterator filter, see lines 21 - 30, listing 8.10.

The remainder of this method creates the `Slab` instance, wraps it in a `CadSlab` proxy component and creates a `Binder` and a `SlabUpdater` instance.

The `SlabUpdater` is responsible for the 'recording' of CAG events that modify the `Slab` instance. At a given point in time during an update, the `SlabUpdater` reassembles the `Slab` instance based on the latest state of the binding geometry and the CAG operations that were defined by the user during the initial construction phase of the `Slab` instance. The `SlabUpdater` is described in detail in section 8.3.5 below.

CAG functionality

Adding shapes to a Slab instance is done using the command class listed and described below.

```
package aho.cademia.sem.cmds;

import aho.cademia.sem.comp.CadSlab;
...
5
public class AddShapeToSlab implements Cmd {

  private CadSlab m_slabProxy;
  private List<Integer> m_indices;
10  private List<NamedCademiaObject> m_wrappedComp
    = new ArrayList<NamedCademiaObject>();
  private SlabUpdater m_updater = null;
  private IBinder m_binder = null;

15  public void doCmd(Object context) throws CmdAbortedException {
    EngModel sem = SemCad.getSEM();
    Kernel krnl = (Kernel) context;
    Database db = krnl.getDatabase();
    NameSpace ns = db.getNameSpace();
20    Set selSet = db.getSelectSet();

    m_indices = new ArrayList<Integer>();

    Iterator iter = CollUtilities.filterableIterator(
25      db.getComponentSet().iterator(),new ClassFilter(CadSlab.class));
    if(iter.hasNext()){
      m_slabProxy = (CadSlab)iter.next();
    }
    else{
30      System.err.println("No slab exists...  [error]");
      return;
    }
    if(iter.hasNext()){
      System.err.println("More than one slab exists...   [error]");
35      return;
    }
```

```
    Iterator<IBinder> _bIter = SemCad.getSEM().getBinders().iterator();
    while (_bIter.hasNext()) {
      m_binder = _bIter.next();
      if(((INamedObject)m_binder.getBoundObject()).getName().
          equals(m_slabProxy.getEngComp().getName())){
        m_updater = (SlabUpdater)m_binder.getUpdater();
        break;
      }
    }
    if(m_updater == null)
      throw new RuntimeException("no updater found!");

    iter = CollUtilities.filterableIterator(selSet.iterator(),
        new ClassFilter(Component.class));

    while (iter.hasNext()) {
      Component comp = (Component) iter.next();
      NamedCademiaObject _ncadObj = null;
      if(comp instanceof CadProxy)
        continue;
      _ncadObj = SemCad.wrap(ns.getName(comp),comp);
      m_wrappedComp.add(_ncadObj);
      sem.add(_ncadObj);

      Shape _shape = comp.getShape(0);
      int index = m_slabProxy.addShape(_shape);
      m_indices.add(index);
      m_binder.addBindingObject(_ncadObj);
      m_updater.add(_ncadObj.getName(), AreaConstructor.ADD);
    }
  }

  public void redoCmd(Object context) {
    for(int i = 0; i < m_wrappedComp.size(); i++){
      String _name = m_wrappedComp.get(i).getName();
      Shape _shape = m_wrappedComp.get(i).getComponent().getShape(0);
      m_binder.addBindingObject(m_wrappedComp.get(i));
      m_updater.add(_name,AreaConstructor.ADD);
      m_slabProxy.addShape(_shape);
    }
  }

  public void undoCmd(Object context) {
    for(int i = m_wrappedComp.size() -1; i >= 0; i--){
      String _name = m_wrappedComp.get(i).getName();
      m_binder.removeBindingObject(_name);
      m_updater.dropPart(_name);
      m_slabProxy.dropShape(m_indices.get(i));
    }
  }

  public boolean changesState() {
    return true;
  }

  public boolean isUndoable() {
    return true;
  }
}
```

Listing 8.11: CADEMIA CadSlab proxy

public void doCmd(Object context) throws CmdAbortedException: This method retrieves the references of the CADEMIA `Kernel`, the `Database`, the `Namespace` and the *select_set*. The next task is to obtain the reference of the `CadSlab` instance inside the `Database`. This is done by traversing the `Database` with a filterable iterator. This command assumes that only one CadSlab instance exists inside the `Database`, if this is not the case, the command stops. Once the `CadSlab` is found, the `Binder` instance associated with the wrapped `Slab` is obtained by traversing the binder set inside the SEM instance. This is done in order to obtain a reference on the

`SlabUpdater`.

At this point the method has references on the wrapped `Slab`, `Binder` and the `SlabUpdater`. Next, it starts to traverse the *select_set* and, for each component found that is **not** a CAD proxy component, it does the following:

- creates a `NamedCademiaObject` by calling the `SemCad.wrap(String name, Component comp)` method.
- adds the created `NamedCademiaObject` to the SEM model.
- retrieves the shape of the component and adds it to the `CadSlab` instance which forwards it to the wrapped `Slab`.
- adds the created `NamedCademiaObject` to the `Binder` instance.
- adds the name of the `NamedCademiaObject` to the updater as well as the operation that was performed. In this case the shape was added to the `Slab` instance thus the operation **ADD** is associated with the shape.

This command is also undoable, thus it keeps track of changes performed to the CADEMIA `Database` and the SEM instance and reverses the changes when the undo command is executed.

8.3.5 Updaters

Updaters are responsible for assisting the user in restoring consistency between two models. Updaters can either perform automatic changes to the bound objects, or it can prompt a user for intervention during the update process. Thus the update process performs the tasks that can be handled automatically and stops at the steps where the user needs to provide expert knowledge. After the user input is obtained the update process continues until more user input is required.

In the pilot application three updaters are implemented. These updaters perform simple tasks that can be executed without user intervention. They are briefly described below.

PointUpdater

The task of a `PointUpdater` is to update the location of an `IPoint` instance based on the location of a specific control point. This is a one-to-one updater, i.e. the set of source objects inside the IBinder instance contains exactly one object. This implemetation allows the source object to be of either type `NamedCademiaObject`, `IControlableShape` or `IConstructiveAreaGeometry`. If the binding object is of any other type, an exception is thrown to indicate that it was not possible for the updater to perform the update.

```
package aho.cademia.sem.updater;

import aho.bindings.IBinder;
```

```
import aho.bindings.Updater;
import aho.cademia.sem.NamedCademiaObject;
import aho.util.geom.IConstructiveAreaGeometry;
import aho.util.geom.IControlableShape;
import aho.util.geom.IPoint;

public class PointUpdater implements Updater {

  private static final long serialVersionUID = 1L;
  private int m_index;

  public PointUpdater(int index){
    this.m_index = index;
  }

  public void update(IBinder binder) {
    Object src = binder.getBindingObjects().iterator().next();
    IPoint dst = (IPoint)binder.getBoundObject();
    Point2D p2d;
    if(src instanceof NamedCademiaObject){
      p2d = ((NamedCademiaObject)src).getComponent().getControlPoint(m_index);
      dst.setPoint(p2d);
    }
    else if (src instanceof IControlableShape) {
      dst.setPoint(((IControlableShape)src).getPoint(m_index));
    }
    else if (src instanceof IConstructiveAreaGeometry){
      dst.setPoint(((IConstructiveAreaGeometry)src).getPointNo(m_index));
    }
    else {
      throw new RuntimeException("Unsupported binding object found!");
    }
  }
}
```

Listing 8.12: Point updater

The `PointUpdater` stores the index number of the binding control point as an attribute and retrieves the binding control point's coordinates using the appropriate method call, which depends on the type of binding object. It then sets the coordinates of the bound `IPoint` instance to correspond with the values obtained from the binding object.

ShapeUpdater

The `ShapeUpdater` is responsible for updating an `IControlableShape` based on the shape of the binding object. The binding object, in this case, is either an `INamedCademiaObject`, `IConstructiveAreaGeometry` or `IControlableShape` instance. Only one binding object is allowed inside the `IBinder` instance.

The `ShapeUpdater` retrieves the shape from the binding object and assigns the retrieved shape to be the shape of the bound object.

```
package aho.cademia.sem.updater;

import aho.bindings.IBinder;
import aho.bindings.Updater;
import aho.cademia.sem.NamedCademiaObject;
import aho.sem.comp.Slab;
import aho.util.geom.IControlableShape;

public class ShapeUpdater implements Updater {

  private static final long serialVersionUID = 1L;

  public void update(IBinder binder) {
    Object src = binder.getBindingObjects().iterator().next();
    IControlableShape dst = (IControlableShape)binder.getBoundObject();
```

```
      if(src instanceof NamedCademiaObject){
        dst.setShape(((NamedCademiaObject)src).getComponent().getShape(0));
      }
      else if(src instanceof IControlableShape){
        dst.setShape(((IControlableShape)src).getShape());
      }
      else if(src instanceof IConstructiveAreaGeometry){
        dst.setShape(((IConstructiveAreaGeometry)src).getArea());
      }
      else {
        throw new RuntimeException("Unsupported binding object found!");
      }
    }
  }
```

Listing 8.13: Shape updater

SlabUpdater

The `SlabUpdater` contains a list in which the persistent identifiers of the binding components are stored as well as the operation, namely add or subtract. This information is used to reassemble the bound `Slab` instance during the update process. The `add`-method is provided to allow an updater instance to be easily modified. For example, additional CAG operations can be registered and performed on the bound `Slab` instance. Its application can be seen in class `AddShapeToSlab`, listing 8.11 line 66.

```
package aho.cademia.sem.updater;

import aho.bindings.IBinder;
import aho.bindings.Updater;
import aho.cademia.sem.NamedCademiaObject;
import aho.sem.comp.Slab;
import aho.util.INamedObject;

public class SlabUpdater implements Updater {

  private static final long serialVersionUID = 1L;
  private List m_comp = new ArrayList();
  private AffineTransform m_transform = new AffineTransform();

  public void add(String name, int operation){
    m_comp.add(name);
    m_comp.add(operation);
  }

  public void replaceSection(String oldName, String newName){
    for(int i = 0; i < m_comp.size(); i = i+2){
      String _name = (String)m_comp.get(i);
      if(_name.equals(oldName))
        m_comp.set(i, newName);
    }
  }

  public void dropPart(String name){
    int index = m_comp.indexOf(name);
    m_comp.remove(index);
    m_comp.remove(index);
  }

  public void update(IBinder binder) {
    Set _set = binder.getBindingObjects();
    Map<String, INamedObject> _map = new HashMap<String, INamedObject>();
    Iterator<Object> iter = _set.iterator();
    while(iter.hasNext()){
      INamedObject nOjb = (INamedObject)iter.next();
      _map.put(nOjb.getName(), nOjb);
    }
    Object bound = binder.getBoundObject();
    Slab _slab = (Slab)bound;
    _slab.clear();
```

```
45      for(int i = 0; i < m_comp.size(); i = i+2){
          String name = (String)m_comp.get(i);
          int operation = (Integer)m_comp.get(i+1);
          if(operation == IConstructiveAreaGeometry.ADD){
            _slab.add(((NamedCademiaObject)_map.get(name)).
50            getComponent().getShape(0));
          }
          else if(operation == IConstructiveAreaGeometry.SUBTRACT){
            _slab.subtract(((NamedCademiaObject)_map.get(name)).
            getComponent().getShape(0));
55        }
        }
      }
    }
```

Listing 8.14: Slab updater

8.4 Summary

A relatively detailed description of the implementation of the pilot application was presented. It is clear that proposed concepts for linking CAD and FEM models, and supporting consistency accross the link can be realized. Furthermore the specific CAD and FEM applications that were used can be substituted with any other applications which provide the required functionality and a suitable programming interface. The classes of the pilot application provide an excellent starting point for developing a SEM model and tools for such applications.

Chapter 9

Conclusions

Criteria by which the success of techniques for supporting consistency in linked CAD and FEM applications can be measured were described in chapter 2. An evaluation of the proposed integration concepts against the listed criteria is performed, followed by an assessment of the scientific contribution of the dissertation. Proposals for further work to be done in this field are described briefly.

9.1 Evaluating the criteria for success

Each of the criteria is listed, together with a summary of the outcome of the dissertation. The end result is that the criteria for success have been attained:

Domain specific: A separate middleware model was introduced which provides the bridge between different domain specific models. In the case of CAD and FEM integration, the Structural Engineering Model (SEM) model provides an engineer with a model that is completely owned by the structural engineering domain and in which the structural design concept can be expressed. This obviates the need to introduce structural design concepts into the CAD domain, or CAD concepts into the structural design domain, thereby complying with the requirement that the domain models must remain domain specific and independent.

Standalone: The transfer of information between any two models is done by value, i.e. the information is duplicated. In the case of CAD-FEM transfer, the transferred copy belongs to the structural engineer, and he/she alone is responsible for the copied information. The applications of the structural engineering domain do not require the CAD application to perform any further task, neither do the CAD application require the engineering applications to perform its function, i.e. all models can operate in a standalone fashion. However, the proposed

binding mechanism can be used to maintain the consistency of copied information.

Application specification: It has been pointed out that in order to provide a link and support consistency in linked models, the components of the candidate CAD system must be persistently identified and a versioning system must be in place. The persistent names are crucial to maintain the link over different work sessions, and the versioning system allows for the detection of changes, thereby supporting the update mechanism.

Working pilot implemetation: A pilot application was created which successfully proves the concepts for integrating CAD and FEM models, with effective support of updating.

Application independent: An analysis of the pilot application, based on its description presented in chapter 8, shows that the proposed solution can be applied to general CAD and FEM applications that comply with the specifications mentioned above.

9.2 Scientific contribution of the dissertation

The research described in the dissertation made scientific contributions on three fronts:

9.2.1 Binding mechanism

Known mathematical results from the field of graph theory were innovatively mapped to the engineering software domain to support the binding and updating mechanism that was proposed. Implementation examples in the CAD-FEM domains were provided to prove the integrity, functionality and effectiveness of the proposed concept. The concept can be applied to link specialized models from other domains as well.

9.2.2 Re-using existing applications

The proxy component design pattern can encourage the development of new specialized domain applications by exploiting any beneficial functionality of existing applications. The design pattern clearly separates the different domain specific models, preventing contamination of any of the models with elements of the other and leaving them free to function in a standalone way.

9.2.3 Using object technology to map solutions to the computer

Throughout the dissertation Java source code is provided to elucidate the implementation of proposed concepts. Much of the source code can be con-

verted and applied to map elements of civil engineering solutions to the computer.

9.3 Future research

9.3.1 Use of middleware models

The development of middleware models between the CAD and construction scheduling domains may lead to new approaches in which 4D models can be spawned without imposing any prerequisites on the person that is responsible for the CAD model. Currently, the creation of 4D models requires a finer grained geometric description than that in which 3D models are typically created [Tulke and Hanff 2007].

A middleware model provides the ideal place where a planner can select objects and adjust the granularity of the objects to suit his/her needs and specify other, domain related information. No expert CAD knowledge is required to select a component and subdivide the component if appropriate software tools are provided.

In this senario, a project planner will have a separate application which allows him to specify the scheduling model based on an imported CAD model. Consistency support can be provided in exactly the same way as described in this dissertation.

This methodology supports the concept that each model performs a specific task and no restrictions are placed on the owner of the model. I.e. the owner of a model does not have to adhere to rules and/or perform tasks which offer no advantage to his model and are only specified to facilitate the work of other parties.

9.3.2 Server-side management of project information

The pilot application, structures@CADEMIA, uses a separate CADEMIA instance to provide persistently identified and versioned geometry. In a commercial environment a project server is required to version, store and manage the objects effectively. This technology was introduced by [Firmenich 2002] and developed further by [Beer 2006].

Once this technology is fully operational, a SEM application can obtain persistently named and versioned objects from a project server and use the project server as repository for its components. This will enable computer supported collaborative work (CSCW), i.e. it will support different structural engineers working together, synchronously on the same project task.

Appendices

Appendix A

Pilot implementation: Step-by-step example

A.1 Start structures@CADEMIA

CADEMIA is started with the following command in a console:

```
java -classpath aho.jar cib.cad.Instance -icademia.ini
```

A.2 Import geometry

The first step is to import the geometry that forms the basis for the analysis that is to be performed. If an instance of the *rmiregistry* is not running on the computer, it must be started before the import can be performed. This is done by typing the following command in a console[1].

```
rmiregistry
```

The *rmiregistry* allows an object to obtain a reference on a remote object (i.e. an object located inside another Java Virtual Machine (JVM) instance). This enables the transfer of data between different Java Virtual Machines via the invocation of methods on the remote objects. The *rmiregistry* is the place where remote objects are registered, and references to these objects are obtained at the *rmiregistry*.

Once the rmiregistry is running, the *import* command can be executed. Another CADEMIA instance appears. The new CADEMIA instance registers itself with the *rmiregistry* and thus allows structures@CADEMIA to establish a reference to it. The new CADEMIA instance is providing named and versioned geometry to structures@CADEMIA. In this application the user can either draw geometry using the CAD functionality provided by

[1] Ensure that the system path finds the rmiregistry command which is located inside the JRE/bin folder and that no classpath is set before this command is executed

CADEMIA, or load an existing CADEMIA file. For the purposes of the example, the floor slab shown in figure A.1 is used in the description that follows.

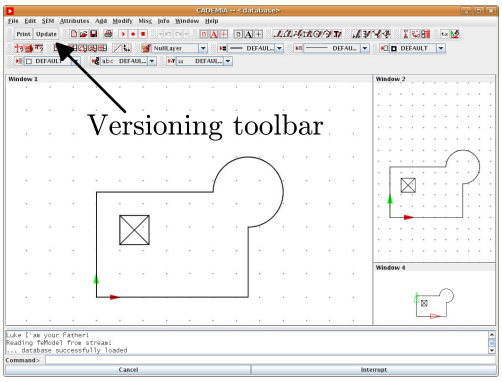


Fig. A.1: Geometry definition on server side using CADEMIA as server

Once the geometry is completed, the *update* command of the versioning toolbar is executed. This assigns version numbers to the geometry. The user now selects the geometry that is to be imported by structures@CADEMIA and returns to structures@CADEMIA where he presses the *getselect* button on the import toolbar. The selected geometry is then imported into structures@CADEMIA and the engineer starts with the definition of the SEM instance.

A.3 Interpret geometry

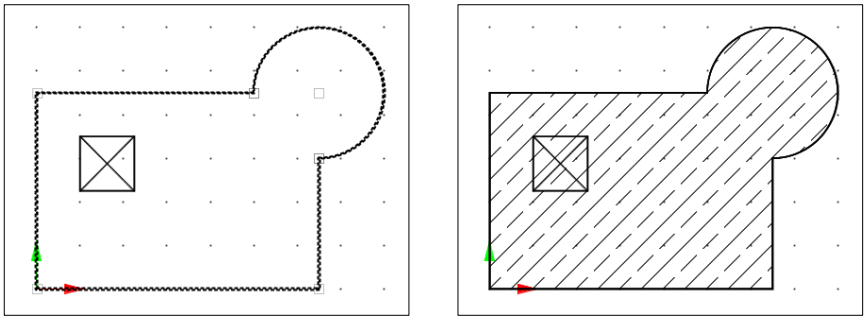


Fig. A.2: Defining the outer boundary of the slab component

The engineer assigns structural semantics to the imported geometry. He first selects the polyline and the arc which forms the outline of the slab shown in figure A.2, left-hand side, and then executes the *Create slab* command from the SEM menu. A slab component and a binder instance is created that

binds the polyline and the arc to the slab component as shown in figure A.2, right-hand side.

Next, he unselects all the selected components and selects the square that represents the opening in the slab. After the square has been selected the *Subtract shapes from slab* command is executed and the opening appears inside the slab at the same position where the square is located. The binder that models the geometry/slab dependency is updated in the background, i.e. the additional dependency between the slab and the square is added to the binder. The result is shown in figure A.3.

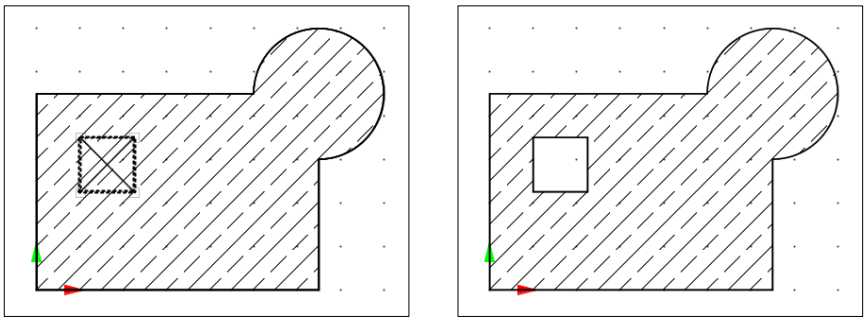


Fig. A.3: Adding the hole to the slab component

The *dropimp* command is executed. The result is that the imported geometry is removed from the runtime instance. The next step is to add four supports to the slab. This is done by selecting the slab to make the contol points of the slab visible. The engineer selects the four control points that defines that main rectangle of the slab. He then issues the *Point support* command of the SEM menu. Four point supports are added to the SEM instance. These four supports are bound to the corresponding control points of the slab as shown in figure A.4)

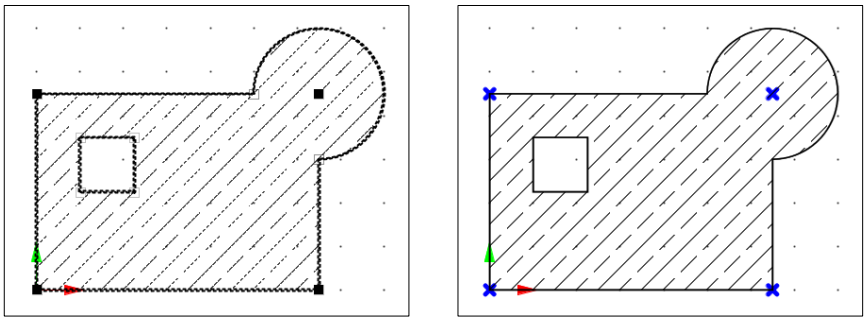


Fig. A.4: Defining the supports of the slab

In order to keep the complexity of the example low, no additional SEM components are created.

A.4 Derive FEM analysis

After finishing the development of the SEM instance, a FEM analysis of the model can be performed. The *Evalute* command of the SEM menu is executed. A file name and a mesh index is required as input. The FEM input file is created and analysed and the results of the model are displayed in a viewer application. The self-weight of the slab is the only load that is applied during the FEM analysis.

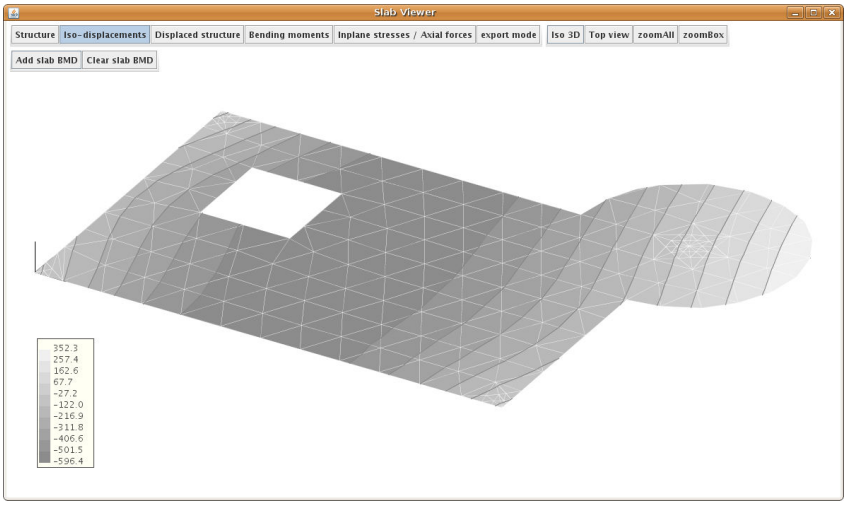


Fig. A.5: Result of first analysis.

A.5 Update SEM instance

To demonstrate the updating functionality, certain geometry changes are applied to the original geometry. A re-import of the geometry is performed. The new versions of the original geometry is displayed, and the SEM components and the changes are visible. The *check* command of the binder toolbar yields the following text output.

```
----------------------
Was Changed : obj1@cademia   obj2@cademia   obj0@cademia
----------------------
Update sequence :
   obj0
   obj4   obj1   obj2   obj3
----------------------
```

The changes to the binding geometry are detected via the new version numbers and the update sequence is calculated and displayed. The slab is

updated first, then the four point supports.

The actual update is performed by executing the *update* command of the binder toolbar.

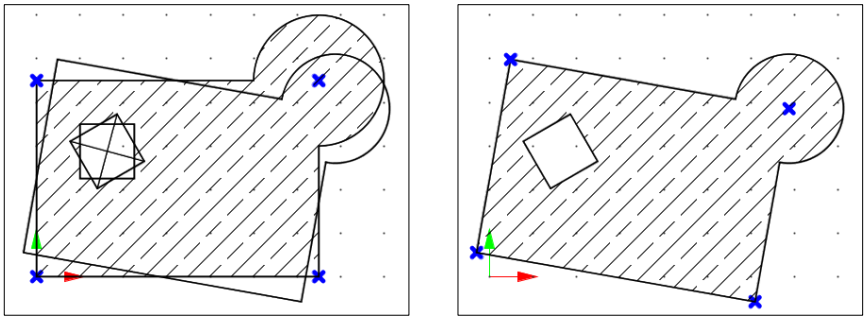


Fig. A.6: Updating the slab

A.6 Re-analyse

After the update the engineer executes the *Evalute* command. A new input file is created, analysed and the results are displayed.

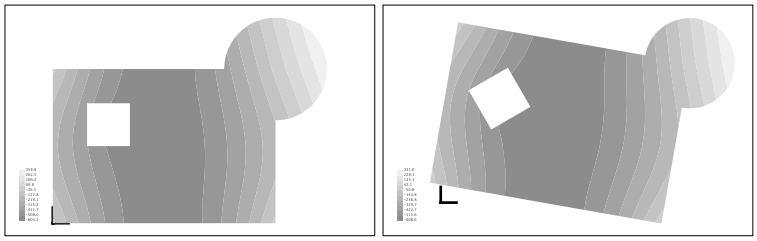


Fig. A.7: Result with updated layout

A.7 Summary

The example gives an indication of how the solution approach would be used in practice. The work done by the structural engineer on the SEM instance, for example the supports that were created, is not lost after the changes to the geometry model were imported. This example clearly shows that it is possible, with little additional effort by the structural engineer, to support the updating of a SEM instance based on changes to binding geometry. Furthermore, because the FEM analysis is derived directly from

the SEM instance, the effect of changes to binding geometry can be obtained with minimal effort.

Appendix B

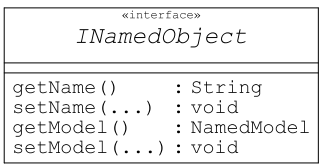
FEM framework

This appendix gives a brief overview of the current state of the framework for finite element analysis that is used in the dissertation as the FEM application. The first version of the framework is described in [Olivier 2002]. The basic framework has remained the same, with the addition of new element- and load types. A short summary of further development of the framework is presented as well.

B.1 Design overview

The following design guidelines are followed throughout the development of the framework:

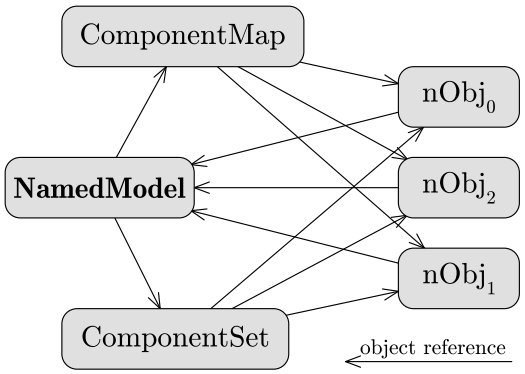
Components: Each component has a unique name. Names are assigned at the instantiation time of the component and are persistent. The name functionality is provided by implementation of the `INamedObject` interface. In addition, each component has a reference on the model instance that contains it. A component may only belong to one model at a time.



Model: A model instance contains a set of components, i.e. `componentSet`. The use of an `IteratorFilter` allows dynamic access to subsets of the `componentSet` based on a specific criterium. For example, when the global stiffness matrix, $[K_s]$ is assembled, an `IteratorFilter` returns all the `IElement` instances contained inside the model. These `IElements` are then used to assemble $[K_s]$.

The model contains a map, i.e. `componentMap`, were each component inside the model is registered using its name as key.

The model instance allows the registation of listeners to notify events that take place on the model. For example if the model is analysed or a component is added/changed/removed, the listener system notifies all registered listeners.



The model also manages the different load sets that exist for a specific FEM instance.

Component dependencies: The `componentMap` inside the model acts as a registry where a component obtains temporary references to related components. No direct inter-component references are allowed between individual components. A component only contains the names of the components on which it depends. When a component needs information from other components, it obtains the runtime references by invoking the `public INamedObject get(String name)` method of the model that contains it, retrieves the data and then discards the reference.

This design ensures that the complexity of the framework remains low, especially if multiple models exist simultaneously in one virtual machine. It also enables the exchange of indiviual components without burdening the framework with the updating of references to the original component when it is replaced by an alternative component. For example, two engineers, in different locations, that want to work on the same FEM instance[1] in real time need to exchange components. If the one changes a component, the changed component is sent to the other engineer and the model is updated immediately to be consistent with the model where the change originated.

[1] In this scenario, two JVM's exists, the model under discussion is loaded in both virtual machines and a peer-to-peer connection exists between the two VM's to share changed objects.

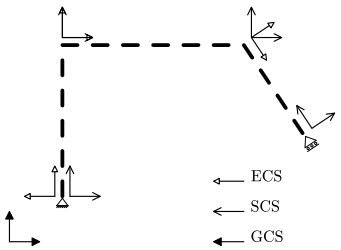
Interfaces: Well defined interfaces describe the functionality of all the implemented FEM components. This makes it possible to add new element types to the finite element framework without influencing the existing elements. Existing abstract classes simplify the implementation of new element types.

B.2 Coordinate systems

Three coordinate systems exist in the framework. The first coordinate system is the geometry coordinate system (**GCS**) which is used to define the geometry of the problem domain. The basis vectors of this coordinate system are defined as follows: $e_1 = \{1, 0, 0\}^T$, $e_2 = \{0, 1, 0\}^T$ and $e_3 = \{0, 0, 1\}^T$.

The second coordinate system is the element coordinate system (**ECS**). The orientation of this coordinate system relative to the **GCS** varies from element to element according to the orientation of the element relative to the **GCS**. The shape functions of elements are typically defined in the ECS and all element integrals are solved inside this coordinate system. The element stiffness matrix $[K_e]$ is computed relative to this coordinate system.

The third coordinate system is the system coordinate system (**SCS**). This coordinate system is the coordinate system in which $[K_s]$ is defined, thus it defines the directions of the displacements at each node. In general the **SCS** coincide with the **GCS**, but it allows the specification of local coordinate systems at specific nodes.

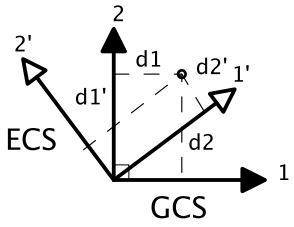


The figure above displays the three coordinate systems. Note that at the node on the right-hand side, the **SCS** does not coincide with the **GCS** in order to enable the definition of a skew support.

B.2.1 Coordinate transformations

The basis vectors of both the **ECS** and the **SCS** are expressed in the **GCS**. The matrix that transforms a vector from the **ECS** to the **GCS** is called the rotation matrix.

$$\{V_{GCS}\} = [R_{ecs}] \times \{V_{ECS}\} \tag{B.1}$$



The first column of $[R]$ contains the coordinates of the first basis vector of the **ECS** expressed in the **GCS**, likewise the second and third columns of $[R]$ contain the coordinates of the second and third basis vectors of the **ECS**.

$$\left\{ \begin{array}{c} d1 \\ d2 \end{array} \right\} = \left[\begin{array}{cc} a & c \\ b & d \end{array} \right] \times \left\{ \begin{array}{c} d1' \\ d2' \end{array} \right\} \tag{B.2}$$

where

$$a = \frac{v1' \cdot e_1}{|v1'|}$$
$$b = \frac{v1' \cdot e_2}{|v1'|}$$
$$c = \frac{v2' \cdot e_1}{|v2'|}$$
$$d = \frac{v2' \cdot e_2}{|v2'|}$$

If a vector is defined in the **GCS** multiplying it with $[R]^T$ transforms it to the **ECS**.

$$\{V_{ECS}\} = [R_{ecs}]^T \times \{V_{GCS}\} \tag{B.3}$$

The transformation from the SCS to the GCS is exactly the same as the transformation described above.

$$\{V_{GCS}\} = [R_{scs}] \times \{V_{SCS}\} \tag{B.4}$$
$$\{V_{SCS}\} = [R_{scs}]^T \times \{V_{GCS}\} \tag{B.5}$$

B.3 Degrees-of-freedom

The framework currently supports only displacement based finite elements with up to six degrees-of-freedom per node. A unique aspect of the framework is that the engineer does not need to specify the analysis domain, e.g.

2D truss or 3D frame, when he/she starts off with the specification of a new FEM instance, as is typically required in existing commercial finite element analysis software.

The framework allows the use of 2D truss elements[2] in a 3D model. For example, to analyse a structural steel roof in 3D, 2D trusses and girders can be utilized. The elements themselves activate the nodal degrees-of-freedom according to their orientation. A 2D truss element activates only two degees-of-freedom at each node in the plane in which it lies. This is done by invoking the `public void setActiveDOFs(byte reqDof)` method specified by the `INode` interface.

Local coordinate systems enable the engineer to specify a different coordinate system for the primal values at a node.

B.4 Elements

All implemented elements extend an abstract class `Element`. This class takes care of the node names, material and the element coordinate system.

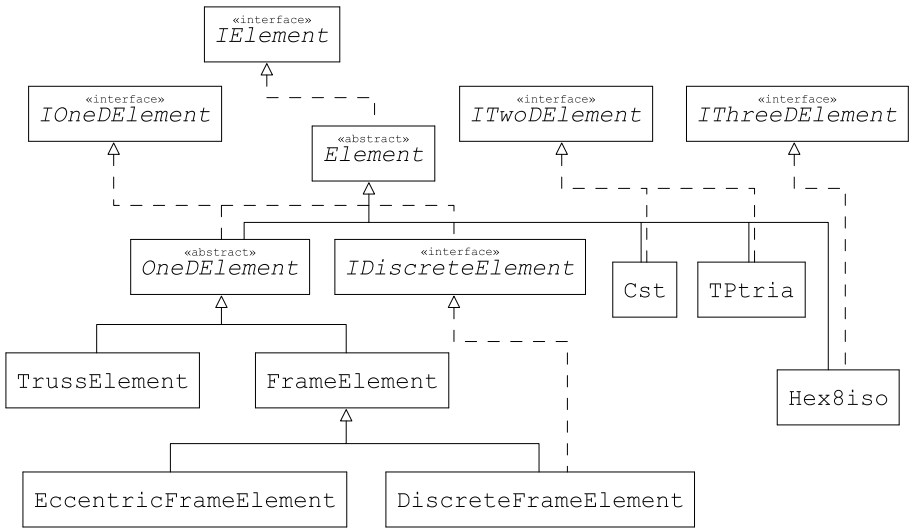
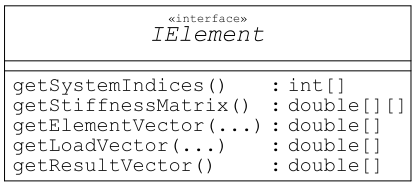


Fig. B.1: Implemented elements

B.4.1 IElement

The `IElement` interface prescribes all the functionality that an element implementation must adhere to in order to be utilized in the framework. Some of the methods specified by the interface are now explained in more detail.

[2] A 2D truss element has 2 displacement degrees of freedom at each node




```
int[] getSystemIndices();
```

This method retrieves the degree-of-freedom indices from the element's nodes and returns the indices as an array. The method that assembles the system stiffness matrix, $[K_s]$ invokes this method in order to correctly insert the element stiffness matrix $[K_e]$ into the system matrix $[K_s]$.

```
double[][] getCoordinateSystem();
```

This method returns the basis vectors of the **ECS** relative to the **GCS**.

```
double[][] getRotationMatrix();
```

This method returns a matrix that transforms the degree-of-freedom displacements from the **ECS** to the **SCS**. This matrix is built up by assembling the different nodal coordinate systems into an element rotation matrix. The general form of this matrix is presented below:

$$[R_{elem}] = \begin{bmatrix} [R]_1 & 0 & 0 & 0 \\ 0 & [R]_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & [R]_n \end{bmatrix} \tag{B.6}$$

where

$$[R]_i = [R_{ecs}]_i \times [R_{scs}]_i^T \tag{B.7}$$

```
double[][] getSystemMatrix();
```

This method returns the stiffness matrix, $[K_e]$, of an element in the **SCS**. Firstly the element matrix is calculated relative to the **ECS** using the following equation:

$$[K_e]_{ecs} = \int_v [S_z][E][S_z]^T dV \tag{B.8}$$

where $[S_z]$ is the strain-displacement matrix and $[E]$ is the material matrix. This integration is either done analytically or numerically using Gauß integration.

The second step is to transform the $[K_e]_{ecs}$ to $[K_e]_{scs}$. This is done using the following equation:

$$[K_e]_{scs} = [R_{ecs}] \times [K_e]_{ecs} \times [R_ecs]^T \tag{B.9}$$

```
double[] getElementVector(double acceleration, double[] globalDir);
```

This method returns a volume load e.g. own weight.

$$[w_e]_{ecs} = \int_v [S][a\rho]dV \tag{B.10}$$

where a =acceleration and ρ = material density.

The element vector is transformed from the **ECS** to the **SCS** with the following equation:

$$[w_e]_{scs} = [R_{ecs}][w_e]_{ecs} \tag{B.11}$$

```
double[] getLoadVector(ILoad load);
```

This method returns a load vector of a load acting on the element based on the interpolation functions of the element itself. It throws a `LoadNotSupported` exception if the type of load is not supported by the element, e.g. `PointLoad` acting perpendicular to a `TrussElement`.

```
double[] getElementResultVector();
```

For 1D elements the method returns the end forces of the element. For 2D or 3D elements the stresses and strain values at specific points inside the element, e.g. the midpoint or the Gauß points, are returned.

B.4.2 IDiscreteElement

The finite element framework allows the use of discrete elements, i.e. bars and beams, combined with standard finite elements.

```
void addLoad(IDiscreteElementLoad load);

void incrementFef(String loadName, String loadSetName, double[]
values);

void reset();
```

B.5 Loads

The following loads are currently available in the framework:

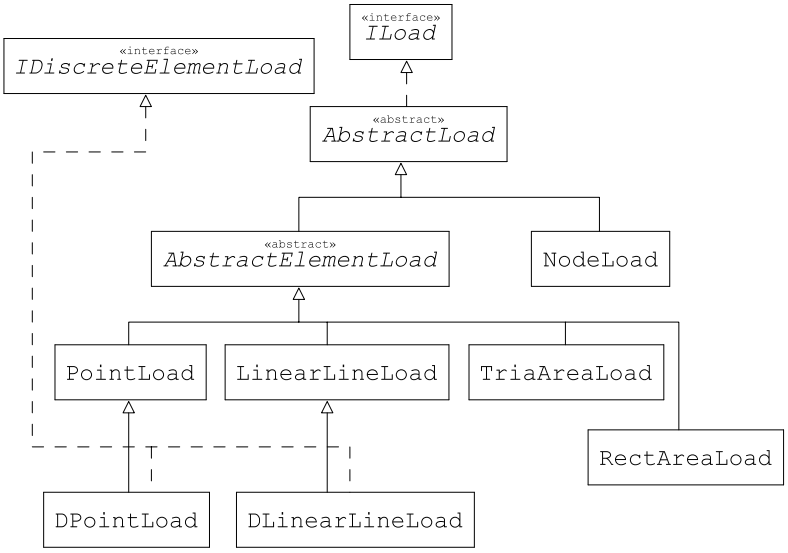


Fig. B.2: Implemented loads

A load contains a direction vector and an intensity. Loads are divided into two main categories, namely element loads and nodal loads.

Nodal loads are applied at the system nodes. These load values are directly inserted into the system load vector during the analysis process. A nodal load is either specified in the **GCS** or the **SCS**.

Element loads, on the other hand, are added to the system load vector after computing the following integral:

$$w_s = \int_b [S]\{W_e\} \tag{B.12}$$

Element loads are either defined relative to the **ECS** or the **GCS**.

B.5.1 ILoad

This interface prescribes the functionality required by a load instance to integrate with the rest of the framework. Some of the methods are discussed below.

```
byte getCoordinateSystem();
```

This method returns the coordinate system in which the load is defined, i.e. either the **GCS** or the **ECS**.

A nodal load may either be defined in the **GCS** or in the **SCS** of the node where it is applied. Note that the **SCS** is the same as the **GCS** if a local coordinate system is not present at the particular node.

```
double[] getDirection(double[] direction, byte cs);
```

This method returns the direction vector of the load. The *cs* parameter specifies the coordinate system in which the direction vector is to be expressed.

```
double[] getLoadVector(byte coordinateSystem);
```

This method returns the load vector of the load instance relative to the given coordinate system.

B.6 Constrain conditions

Constraint equations are used to express certain degrees-of-freedom displacements in terms of others.

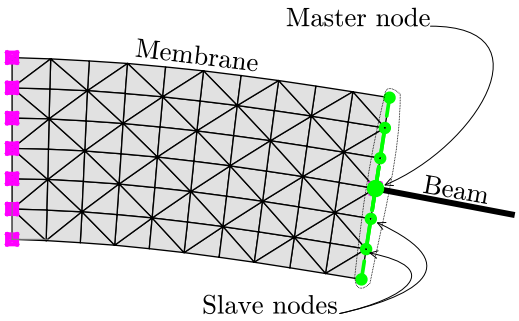


Fig. B.3: Constraint application example

The figure above shows an example of the application of constraint conditions. The displacement of the slave nodes are not solved during the solution of the system of equations. These slave degrees of freedom are removed from the system, by expressing them in terms of the degrees-of-freedom of the master node.

The following two equations describe the slave degrees-of-freedom in terms of the master degrees-of-freedom:

$$x_s = x_m + Z_m \times L \tag{B.13}$$

where x_s is the x-translation of the slave node, likewise x_m the x-translation dof of the master node, Z_m the z-rotation of the master node and L is the y-coordinate of the master node minus the y-coordinate of the slave node.

$$y_s = y_m \tag{B.14}$$

where y_s is the y-translation of the slave node and y_m the y-translation of the master node.

A compatibility matrix, $[A]$ is defined that contains all the constraint equations. The number of rows of this matrix is equal to the number of master degrees-of-freedom, i.e. the number of system equations to be solved. The number of columns of $[A]$ is equal to the total number of degrees-of-freedom of the system.

The following two equations reduces the dimension of the system stiffness matrix and the load vectors:

$$[K_s]^* = [A] \times [K_s] \times [A]^T \tag{B.15}$$

$$[W_s]^* = [A] \times [W_s] \tag{B.16}$$

The system is then solved using the following equation:

$$[K_s]^* \{U_s\}^* = [W_s]^* \tag{B.17}$$

and $\{U_s\}$ is calculated by:

$$\{U_s\} = [A]^T \{U_s\}^* \tag{B.18}$$

Appendix C

Topological sorting: Performance comparison

Vertices	Edges							
	10	50	250	500	1000	2000	5000	10000
100	1	9	24	34	77	92	-	-
250	2	12	27	49	89	130	229	423
500	4	15	32	63	114	203	382	631
1000	9	19	46	99	154	258	570	1214
2500	22	40	96	128	207	368	1104	2499
5000	33	54	111	190	261	587	1656	3899
7500	36	55	160	240	347	669	1870	5909

Table C.1: AHO : Time usage [sec]

Vertices	Edges							
	10	50	250	500	1000	2000	5000	10000
100	4	5	25	35	39	42	-	-
250	23	26	40	51	109	170	202	180
500	65	68	106	96	104	172	857	1041
1000	129	140	104	126	126	304	2859	5972
2500	424	408	401	417	474	404	725	17537
5000	-	-	-	-	-	-	-	-
7500	-	-	-	-	-	-	-	-

Table C.2: IKM : Time usage [sec]

Vertices	Edges							
	10	50	250	500	1000	2000	5000	10000
100	4.0	0.6	1.0	1.0	0.5	0.5	-	-
250	11.5	2.2	1.5	1.0	1.2	1.3	0.9	0.4
500	16.2	4.5	3.3	1.5	0.9	0.8	2.2	1.6
1000	14.3	7.4	2.3	1.3	0.8	1.2	5.0	4.9
2500	19.3	10.2	4.2	3.3	2.3	1.1	0.7	7.0
5000	-	-	-	-	-	-	-	-
7500	-	-	-	-	-	-	-	-

Table C.3: IKM/AHO : Time usage ratio

Vertices	Edges							
	10	50	250	500	1000	2000	5000	10000
100	0	0	62	71	191	365	-	-
250	0	0	62	66	197	341	804	1615
500	0	0	62	128	198	389	898	1794
1000	0	0	60	141	197	377	1032	1847
2500	0	83	96	193	282	427	989	1874
5000	145	148	211	279	404	597	1158	1951
7500	205	219	272	341	483	729	1223	2216

Table C.4: AHO : RAM usage [KByte]

Vertices	Edges							
	10	50	250	500	1000	2000	5000	10000
100	62	62	62	62	62	62	-	-
250	311	311	311	311	311	311	311	340
500	1242	1242	1242	1242	1242	1242	1228	1251
1000	3940	3939	3939	3935	3927	3935	3933	4941
2500	23824	24146	24146	24146	24107	24447	24394	24080
5000	-	-	-	-	-	-	-	-
7500	-	-	-	-	-	-	-	-

Table C.5: IKM : RAM usage [KByte]

Vertices	Edges							
	10	50	250	500	1000	2000	5000	10000
100	-	-	1.0	0.9	0.3	0.2	-	-
250	-	-	5.0	4.7	1.6	0.9	0.4	0.2
500	-	-	19.9	9.7	6.3	3.2	1.4	0.7
1000	-	-	65.5	27.9	20.0	10.4	3.8	2.7
2500	-	292.6	252.6	125.0	85.6	57.2	24.7	12.9
5000	-	-	-	-	-	-	-	-
7500	-	-	-	-	-	-	-	-

Table C.6: IKM/AHO : RAM usage ratio

List of Figures

Listings

Bibliography

[Alda et al 2004] Alda, Sascha ; Bilek, Jochen ; Hartmann, Dietrich ; Cremers, Armin B.: Support of Collaborative Structural Design Processes through the Integration of Peer-to-Peer and Multiagent Architectures. In: *Xth International Conference on Computing in Civil and Building Engineering*, 2004

[Autodesk 2007] Autodesk: 2007. – URL www.autodesk.com/revit. – (06.07.2007)

[Beer 2006] Beer, Daniel G.: *Systementwurf für verteilte Applikationen und Modelle im Bauplanungsprozess*, Bauhaus-Universität Weimar, Dissertation, 2006

[Beucke et al 2007] Beucke, K. ; Firmenich, B. ; Beer, D. G. ; Richter, T.: *Entwurf und Verifizierung einer CAD-Systemarchitectur zur Unterstützung der verteilten technischen Bearbeitung im Konstruktiven Ingenieurbau.* In: Rüppel, U (Editor): *Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau*, Springer, 2007

[Bilchuk 2005] Bilchuk, Irina: *Generalisierte Informationsstrukturen für Anwendungen im Bauwesen*, TU Berlin, Dissertation, 2005

[Bray et al 2006] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M. ; Maler, Eve ; Yergeau, François: *Extensible Markup Language (XML) 1.0 (Fourth Edition) - Origin and Goals. World Wide Web Consortium.* September 2006. – URL http://www.w3.org/TR/2006/REC-xml-20060816/. – (14.08.2007)

[CADEMIA 2007] CADEMIA: 2007. – URL www.cademia.org. – (14.08.2007)

[Date 2000] Date, C. J.: *An introduction to Database Systems.* 7th edition. Addison-Wesley, 2000. – ISBN 0-201-68419-5

[DFG-SPP 2007] DFG-SPP: *DFG-Schwerpunktprogramm 1103 Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau.* 2007. – URL www.dfg-spp1103.de. – (08.13.2007)

[Eastman 1999] Eastman, Charles M.: *Building Product Models: Computer Environments Supporting Design and Construction.* CRC Press, 1999. – ISBN 0-8493-0259-5

[ETABS 2007] ETABS: 2007. – URL www.csiberkeley.com/products_ETABS. – (06.08.2007)

[Firmenich et al 2005] Firmenich, B. ; Koch, C. ; Richter, T. ; Beer, D. G.: Versioning structured object sets using text based Version Control Systems. In: *CIB W78 22nd Conference on Information Technology in Construction, Dresden*, 2005

[Firmenich 2002] Firmenich, Berthold: *CAD im Bauplanungsprozess: Verteilte Bearbeitung einer strukturierten Menge von Objektversionen*, Bauhaus-Universität Weimar, Dissertation, 2002

[Firmenich 2006] Firmenich, Berthold: *Vorlesungen über CAD in der Bauinformatik: Grundlagenorientierter Systementwurf.* 2006

[Hanff 2003] Hanff, Jochen: *Abhängigkeiten zwischen Objekten in ingenieurwissenschaflichen Anwendungen*, TU Berlin, Dissertation, 2003

[IAI 2007] IAI: 2007. – URL www.iai-international.org. – (06.07.2007)

[IFC 2007] IFC: 2007. – URL http://www.ifcwiki.org/index.php/Main_Page. – (06.07.2007)

[Java 2007] Java, API: *Java™, Standard Edition 6 API Specification.* 2007. – URL http://java.sun.com/javase/6/docs/api/. – (14.08.2007)

[Katzenbach et al 2006] Katzenbach, Rolf ; Rüppel, Uwe ; Meissner, Udo F. ; Giere, Johannes ; Wagenknecht, Armin: A Process-oriented Cooperation Model for distributed Civil Engineering Construction Works. In: *XIth International Conference on Computing in Civil and Building Engineering*, 2006, p. 3266–3275

[Klinger et al 2006] Klinger, Axel ; Berkhahn, Volker ; König, Markus: Formal treatment of additions in Planning Processes. In: *XIth International Conference on Computing in Civil and Building Engineering*, 2006, p. 2883–2891

[Koch and Firmenich 2006] Koch, C. ; Firmenich, B.: Operative models for the introduction of additional semantics into the cooperative planning process. In: *Proceedings of the 13th EG-ICE Workshop "Intelligent Computing in Engineering and Architecture"*, June 2006

[König 2004] König, M.: *Ein Prozessmodell für die kooperative Gebäudeplanung*, Dissertation, 2004

[Kraft and Nagl 2007] Kraft, Bodo ; Nagl, Manfried: Graphbasierte Werkzeuge zur Unterstützung des konzeptuellen Gebäudeentwurfs. In: Rüppel, Uwe (Editor): *Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau*, Springer, 2007

[Kraft and Retkowitz 2006] Kraft, Bodo ; Retkowitz, Daniel: Rule-Dependencies for visual knowledge Specification in conceptual design. In: *XIth International Conference on Computing in Civil and Building Engineering*, 2006

[Liebich and Wix 2000] Liebich, Thomas ; Wix, Jeffrey: IFC Technical Guide / International Alliance for Interoperability. 2000. – Research Report

[Olivier 2002] Olivier, A.H.: *Object-Oriented Finite Element Framework*, University of Stellenbosch, Master thesis, 2002

[Pahl and Damrath 2001] Pahl, P. J. ; Damrath, R: *Mathematical Foundations of Computational Engineering.* Springer, 2001. – 574–579 p. – ISBN 3-540-67995-2

[Pahl and Beucke 2000] Pahl, P.J. ; Beucke, K.: Neuere Konzepte des CAD im Bauwesen: Stand und Entwicklungen. In: *Internationales Kolloquium über Anwendungen der Informatik und Mathematik in Architektur und Bauwesen (IKM)*, 2000

[Perevalova and Pahl 2004] Perevalova, J. ; Pahl, P. J.: Structural and Functional Dependence of Objects in Data Bases. In: *Xth International Conference on Computing in Civil and Building Engineering* ICCCBE-X (event), 2004

[Rank et al 2007] Rank, E. ; Romberg, R ; Niggl, A: *Volumenorientierte Modellierung als Grundlage einer vernetzt-kooperativen Planung im Konstruktiven Ingenieurbau.* In: Rüppel, U (Editor): *Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau*, Springer, 2007

[Romberg et al 2004] Romberg, R. ; Niggl, A. ; Treeck, C. van ; Rank, E.: Structural Analysis based on the Product Model Standard IFC. In: *Xth International Conference on Computing in Civil and Building Engineering*, 2004

[Rüppel 2007] Rüppel: *Grundlegende Betrachtungen zur vernetzten Kooperation.* In: Rüppel, U (Editor): *Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau*, Springer, 2007

[Rüppel and Lange 2006] Rüppel, U ; Lange, M: An integrative process model for cooperation support in civil engineering. In: *Electronic Journal of Information Technology in Construction* ITcon Vol. 11 (2006), p. 509–528. – URL http://www.itcon.org/2006/37. – (20.07.2007)

[Schnellenbach-Held et al 2004] Schnellenbach-Held, Martina ; Hartmann, Markus ; Pullmann, Torben: Knowledge Based Systems in Distributed Design Environments. In: *Xth International Conference on Computing in Civil and Building Engineering*, 2004

[Senescu et al 2006] Senescu, Reid ; Mole, Andrew ; Fresquez, Anthony: A cade study in structural drafting, analysis and design using an integrated intelligent model. In: *Joint International Conference on Computing and Decision Making in Civil and Building Engineering*, 2006, p. 1797–1806

[Tauscher et al 2007] Tauscher, Eike ; Mikulakova, Eva ; König, Markus ; Beucke, Karl: Generating Construction Schedules with Case-Based Reasoning Support. In: *Computing in Civil Engineering*, ASCE, 2007, p. 119–126

[Tulke and Hanff 2007] Tulke, Jan ; Hanff, Jochen: 4D Construction Sequence Planning - new process and data model. In: *CIB W78 24th Conference on Information Technology in Construction, Maribor*, 2007, p. 79–84

[Turau 1996] Turau, Volker: *Algorithmische Graphentheorie*. Addison-Wesley, 1996. – ISBN 3-89319-938-1

[van Rooyen 2002] van Rooyen, Gert C.: *Structural analysis in a distributed collaboratory*, University of Stellenbosch, Dissertation, 2002

[Wikipedia 2007a] Wikipedia: 2007. – URL http://en.wikipedia.org/wiki/AutoCAD_DXF. – (15.05.2007)

[Wikipedia 2007b] Wikipedia: 2007. – URL http://en.wikipedia.org/wiki/Software_agents. – (15.05.2007)